Implementierung und Test des ESA LEON VHDL-Cores - Entwicklung eines PMC-Boards

Diplomarbeit

von
Dominik Mader

Diplomarbeit

von
Dominik Mader
Matr.-Nr.: 56910

Fachhochschule:	Hochschule Bremen
Fachbereich:	Elektrotechnik
Fachrichtung:	Informationstechnik
Betreuender Betrieb:	Astrium GmbH
Betreuer:	Dipl. Ing. Lutz Buttelmann

1. Prüfer:	Prof. Dipl. Ing. Heinz Purnhagen
2. Prüfer:	Prof. Dr. Ing. Sven Simon

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Diplomarbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben habe.

_____

Datum Unterschrift

Inhaltsverzeichnis

1. Einleitung 8
2. Grundlagen 10
   2.1. Hardware 10
      2.1.1. AVNET-Evaluations-Board 10
      2.1.2. PMC-Caddy 12
      2.1.3. PMC-Board 13
   2.2. VHDL-Cores 19
      2.2.1. LEON 19
      2.2.2. AMBA-PCI-Bridge 50
3. Eigenentwicklungen 52
   3.1. PMC-LEON-Board 52
      3.1.1. Anforderungsliste 52
      3.1.2. Problematiken 53
      3.1.3. Bauteile 55
      3.1.4. Schematics und Funktionalität 59
   3.2. PCI-Interrupt Controller 74
      3.2.1. PCI-Interrupt Leitungen 74
      3.2.2. PCI-Interrupt Controller 74
4. Implementierung und Test des LEON auf AVNET-Board 77
   4.1. Implementierung 77
      4.1.1. Konfiguration des LEON 78
      4.1.2. Änderungen am Source-Code 79
   4.2. Test 80
      4.2.1. Testumgebung 81
      4.2.2. Boot Loader 84
      4.2.3. Test der Schnittstellen und zugehöriger LEON-Funktionen 86
      4.2.4. Test der internen LEON-Funktionen 94
   4.3. Zukünftige Tests 97
5. Implementierung und Test des LEON auf PMC-Board 98
   5.1. Test der Bauteile 98
6. Fazit und Ausblicke 100
A-1 Quellenverzeichnis 102
A-2 Schematics 104
A-3 Quellcodes 117

Verzeichnis der Tabellen

Tabelle 2-1 Beziehung zwischen Abstand Host-CMC und Bauteilhöhen 15
Tabelle 2-2 BUSMODE# Kodierung 16
Tabelle 2-3 PMC-Steckverbinder Pn1 / Pn2 17
Tabelle 2-4 PMC-Steckverbinder Pn3 / Pn4 18
Tabelle 2-5 Register-Adressierung und Benennung innerhalb eines Register-Fensters 22
Tabelle 2-6 Trap Table des LEON (Quelle: [L10]) 29
Tabelle 2-7 ASI Gebrauch (Quelle: [L10]) 31
Tabelle 2-8 Standard AHB-Mapping (Quelle: [L10]) 33
Tabelle 2-9 On-Chip Register / APB-Mapping (Quelle: [L10]) 34
Tabelle 2-10 Interrupt Tabelle (Quelle: [L10]) 36
Tabelle 2-11 Vordefinierte Belegung des PIO (Quelle: [L10]) 41
Tabelle 2-12 Adressen-Zuordnung des Memory Controllers (Quelle: [L10]) 41
Tabelle 2-13 AHB-Address-Mapping von DSU und DCL (Quelle: [L10]) 45
Tabelle 2-14 Packages zur Konfiguration 46
Tabelle 3-1 Signalbeschreibung des Top-Levels 62
Tabelle 3-2 Bypass-Kondensatoren für das FPGA 64
Tabelle 3-3 Belegung des Flachbandkabelsteckers 67
Tabelle 3-4 mögliche Widerstandsteiler zur Erzeugung von 3,3V 68
Tabelle 3-5 Jumpereinstellungen zur Konfiguration und Scrubbing 70
Tabelle 3-6 Vergleich der Ethernet PHY Beschaltung 72
Tabelle 4-1 Belegung des PIO-Port 88

Verzeichnis der Abbildungen

Abb. 2-1 AVNET Virtex–E 1000 Evaluation Board (Quelle: [D18]) 10
Abb. 2-2 AVNET Board Blockdiagramm (Quelle: [D18]) 11
Abb. 2-3 PMC-Caddy (Quelle: [D16]) 13
Abb. 2-4 Blockschaltbild des PMC-Caddys (Quelle: [D16]) 13
Abb. 2-5 Typische Single CMC (Quelle: [L7]) 14
Abb. 2-6 CMC auf einer Host Karte (Quelle: [L7]) 15
Abb. 2-7 Einhüllende einer Single CMC (Quelle: [L7]) 16
Abb. 2-8 Blockdiagramm des LEON (Quelle: [L10]) 20
Abb. 2-9 LEON IU Blockdiagramm (Quelle: [L10]) 21
Abb. 2-10 3 überlappende Register-Fenster und die globalen Register (Quelle: [L14]) 22
Abb. 2-11 Allg. Registersatz mit 8 Fenstern (ohne globale Register) (Quelle: [L14]) 24
Abb. 2-12 Processor State Register (PSR) (Quelle: [L10]) 25
Abb. 2-13 Trap Base Register (TBR) (Quelle: [L10]) 26
Abb. 2-14 Watchpoint Register (Quelle: [L10]) 27
Abb. 2-15 Tag-RAM Zeile (Quelle: [L10]) 29
Abb. 2-16 Cache Control Register (CCR) (Quelle: [L10]) 31
Abb. 2-17 Blockdiagramm des Interrupt Controller (Quelle: [L10]) 35
Abb. 2-18 Blockdiagramm der Timer (Quelle: [L10]) 36
Abb. 2-19 Blockdiagramm UART (Quelle: [L10]) 38
Abb. 2-20 Sendeformat der UART (Quelle: [L10]) 38
Abb. 2-21 UART Control Register (Quelle: [L10]) 39
Abb. 2-22 UART Status Register (Quelle: [L10]) 40
Abb. 2-23 I/O Port Interrupt Configuration Register (Quelle: [L10]) 40
Abb. 2-24 Memory Interface des LEON (Quelle: [L10]) 42
Abb. 2-25 DSU und DCL (Quelle: [L10]) 43
Abb. 2-26 Hierarchie LEON 48
Abb. 2-27 Hierarchie des Prozessors 49
Abb. 2-28 Block Diagramm AMBA-PCI-Bridge (Quelle: [L15]) 50
Abb. 3-1 Blockschaltbild Xilinx XC18V00 Serie (Quelle: [D11]) 57
Abb. 3-2 Belegung des FPGAs nach Funktionsgruppen 65
Abb. 3-3 Beispielhaftes PCI-Interrupt Routing 74
Abb. 3-4 Interrupt Kontroll Register der AMBA-PCI-Bridge (irq_reg) 75
Abb. 3-5 Blockschaltbild des PCI Interrupt Controllers 76
Abb. 4-1 Testaufbau 1 82
Abb. 4-2 Testaufbau 2 (UART 2 – Test) 82
Abb. 4-3 Testaufbau 3 (PCI-Test: LEON als Initiator) 83
Abb. 4-4 Testaufbau 4 (PCI-Test: LEON wird adressiert) 83
Abb. 4-5 Programmstruktur des Boot Loaders 85
Abb. 4-6 Programmstruktur „Hello world“ 87
Abb. 4-7 Programmstruktur UART 2-Test 88
Abb. 4-8 Programmstruktur PIO-Test 89
Abb. 4-9 Programmstruktur Flash-Test 92
Abb. 4-10 Programmstruktur für Test des IRQ-Controller / Timer 1 / Timer 2 96

Abkürzungsverzeichnis

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASI	Address Space Identifier
ASR	Ancillary State Register
CCR	Cache Control Register
CMC	Common Mezzanine Card
CMOS	Complementary Metal Oxide Silicon
CP	Co Processor
CPLD	Complex Programable Logic Device
CWP	Current Window Pointer
DCL	Debug support unit Communication Link
DSU	Debug Support Unit
EMV	ElektroMagnetische Verträglichkeit
FME	Fault Managment Element
FPGA	Field Programable Gate Array
FPU	Floating Point Unit
FTC	Fault Tolerant Computer
GUI	Graphical User Interface
icc	integer condition code
IP	Intelectual Property
IRQ	Interrupt ReQuest
ISP	In System Programming
IU	Integer Unit
JTAG	Joint Test Action Group
LED	Light Emitting Diode
LRR	Least Recently Replaced
LRU	Least Recently Used
LVTTL	Low Voltage Transistor-Transistor Logic
MAC	Media Access Control
MII	Media Independent Interface
nPC	next Program Counter
PC	Program Counter
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PHY	PHYsical layer
PIL	Processor Interrupt Level
PIO	Parallel I/O-port
PMC	PCI Mezzanine Card
PROM	Programable Read Only Memory
PSR	Processor State Register
RAM	Random Access Memory
ROM	Read Only Memory
SDRAM	Synchronous Dynamic Random Access Memory
SMD	Surface Mounted Device
SPARC	Scalable Processor ARChitecture
SPLC	Standard PayLoad Computer
SoC	System on Chip

SRAM	Static Random Access Memory
TBA	Trap Base Address
TBR	Trap Base Register
TT	Trap Type
TTL	Transistor-Transistor Logic
UART	Universal Asynchronus Receiver Transmitter
VHDL	Very highspeed integrated circuit Hardware Description Language
VME	Versa Module Eurocard
WIM	Window Invalid Mask register

1. Einleitung

Ziel der vorliegenden Diplomarbeit ist es, ein Demonstrationsboard für einen neuen Prozessortyp zu entwickeln und diesen Prozessor darauf zu testen. Dieser Prozessor ist der ESA LEON, der als VHDL-Core von der Firma Gaisler Research zu Verfügung gestellt wird. Ist die Funktionsfähigkeit des Prozessors erprobt, soll er u. U. als neuer Prozessortyp in verschiedenen Produkten der Firma Astrium ein älteres Modell ablösen. Diese Produkte sind z.B. der *Fault Tolerant Computer* (FTC) oder der *Standard Payload Computer* (SPLC). Da der Prozessor als VHDL-Modell vorliegt, setzt sich die Diplomarbeit aus zwei thematisch unterschiedlichen Aufgabenbereichen zusammen.

Der erste Aufgabenbereich besteht aus der Entwicklung des Demonstrationsboards. Die Planung für den zukünftigen FTC der Firma Astrium sieht vor, diesen als PMC-Board zu entwickeln. Dieser Board-Typ ist eine Mezzanine Card, die rittlings auf einem Trägerboard sitzt und Daten mit diesem über den PCI-Bus (*Peripheral Component Interconnect*) austauscht. Um den LEON in dieser Umgebung testen zu können, wird das Demonstrationsboard als PMC-Board entwickelt. Die Entwicklung umfasst Auswahl der Bauteilgruppen, Erstellung von Schematics und Pinout der programmierbaren Komponenten. Dabei ist sich nach Kriterien zu richten, die durch Astrium und die Spezifikation für PMC-Boards vorgegeben sind. Das Layout übernimmt ein anderer Mitarbeiter.

Der zweite Aufgabenbereich besteht aus der Modifizierung von Source-Code, Implementierung des Designs und anschließendem Test grundlegender Funktionen des Prozessors. Bevor Implementierung und Test auf dem entwickelten PMC-Board durchgeführt werden, soll dies auf einem Evaluierungs-Board der Firma AVNET geschehen. Dadurch werden Fehlerquellen aufgrund des PMC-Designs ausgeschlossen. Für das entwickelte Board müssen gesondert Tests entworfen werden, um die Funktionalität der einzelnen Bauteilgruppen zu verifizieren.

Die Modifizierung des Source-Codes umfasst zwei Teilbereiche. Da PMC-Karten den PCI-Bus benutzen, diese Schnittstelle aber nicht durch das LEON-Design unterstützt wird, muss ein vorhandener VHDL-Core, der eine PCI-Bridge beschreibt, in das LEON-VHDL Modell integriert werden. Der Source-Code der Bridge ist dahingehend zu verändern, dass der LEON einen PCI-Interrupt auslösen kann und PCI-Interrupts an den LEON weitergegeben werden können.

Das Testen des LEONs muss durch die Abarbeitung von Software auf diesem geschehen. Grundlage dafür ist das Verständnis des Prozessors und seiner Komponenten. Aufbauend auf diesem Wissen kann Testsoftware entwickelt werden. Dazu sind Kenntnisse in Assembler und C notwendig.

Durchführung

Der LEON inklusive Bridge ist auf dem Evaluations-Board implementiert und getestet worden. Die Testprogramme können – mit wenigen Modifikationen – zum Testen des entwickelten PMC-Boards verwendet werden. Sie beschäftigen sich zum überwiegenden Teil mit den Schnittstellen zwischen LEON und Peripherie. Sie bestätigen die grundlegende Funktionsfähigkeit des LEON-Designs. Dabei bildet die PCI-Schnittstelle eine Ausnahme: es wurden Fehler bei Übertragungen festgestellt, die auf Timing-Probleme hinweisen. Eine genaue Fehleranalyse und Fehlerbehebung muss in Zukunft durchgeführt werden.

Obwohl die Entwicklung der PMC-Karte vorrangig bearbeitet worden ist, um eine Inbetriebnahme des Boards sowie die Implementierung des LEONs darauf im Rahmen der Diplomarbeit zu ermöglichen, ist zum 31.07.03 das Layout nicht abgeschlossen. Deshalb konnten Implementierung des LEON-Designs und Testergebnisse des Evaluierungs-Boards nicht auf das PMC-Board übertragen werden. Da die folgenden Produktionsschritte mindestens 6 Wochen dauern, ist auf eine Verlängerung der Diplomarbeit verzichtet worden. Die Tests, die die Funktionsfähigkeit der Bauteile des PMC-Boards bestätigen sollen, sind im Rahmen der Diplomarbeit schon festgelegt worden.

Trotz der beim Testen festgestellten Fehler bei Übertragungen zwischen PCI-Bus und LEON zeigen die Ergebnisse dieser Diplomarbeit, dass das getestete Design grundsätzlich funktionsfähig ist. Nach einer Fehlerbehebung und weiteren erfolgreichen Tests, die sämtliche Funktionen des LEON überprüfen, kann das Design als Prozessor in Applikationen zum Einsatz kommen.

Struktur der schriftlichen Ausarbeitung

Die schriftliche Ausarbeitung der Diplomarbeit ist in mehrere Abschnitte unterteilt.

Kapitel 2beschreibt die Grundlagen der verwendeten Hardware, der verwendeten VHDL-Cores und der zur Entwicklung des PMC-Boards benutzten Spezifikationen.

Eigenentwicklungen, die Hardware betreffen, sind in Kapitel 3beschrieben. Darunter fällt neben der Entwicklung des PMC-Boards auch das Handling der PCI-Interrupt-Signale.

In Kapitel 4wird die Implementierung und der Test des LEON auf dem Evaluations-Board beschrieben. Dies umfasst die Beschreibung der Testumgebung, der Testprogramme und aufgetretener Fehler. Außerdem sind Richtlinien für zukünftige Tests beschrieben.

Da das PMC-Board noch nicht vorhanden ist, musste auf eine Implementierung des LEONs auf diesem verzichtet werden. Zukünftige Tests, die die Funktionsfähigkeit der Bauteile des PMC-Boards betreffen, sind in Kapitel 5aufgeführt.

Kapitel 6fasst die Ergebnisse der Diplomarbeit zusammen.

Im Anhang befinden sich das Quellenverzeichnis, die für das Verständnis des entwickelten PMC-Boards notwendigen Schematics und alle Programm-Quellcodes.

2. Grundlagen

In diesem Kapitel sollen die Grundlagen beschrieben werden, die für ein weitergehendes Verständnis notwendig sind. Dabei werden die Bereiche Hardware und VHDL-Cores getrennt betrachtet. Im Kapitel Hardware wird auf die verwendeten Testboards und auf allgemeine Grundlagen des zu entwickelnden PMC-Boards eingegangen. Unter VHDL-Cores wird die Funktionsweise der verwendeten VHDL-Modelle beschrieben. Besonderer Schwerpunkt dabei ist das LEON-Design, das den Prozessor enthält. Neben der allgemeinen Funktionsweise werden auch Spezifikationen betrachtet, um im späteren Verlauf darauf zurückgreifen zu können.

2.1. Hardware

2.1.1. AVNET-Evaluations-Board

2.1.1.1. Beschreibung

Das Evaluations-Board ADS-XLX-VE-DEV (Abb. 2-1), das über die Firma AVNET, Inc. vertrieben wird, ist eine Leiterplatte, die sowohl eine große Anzahl von Schnittstellen, als auch Speicher zu Verfügung stellt. Das „Herzstück“ bildet ein FPGA. Diese Kombination ermöglicht das Testen einer großen Bandbreite von Anwendungen. Das Board eignet sich deshalb auch für die Evaluierung des LEON-Designs, weil die gesamte Peripherie vorhanden ist, die mit dem Design angesteuert werden soll. Im Folgenden wird nur auf die Spezifika des Boards eingegangen, die für die Tests des LEONs benutzt werden.

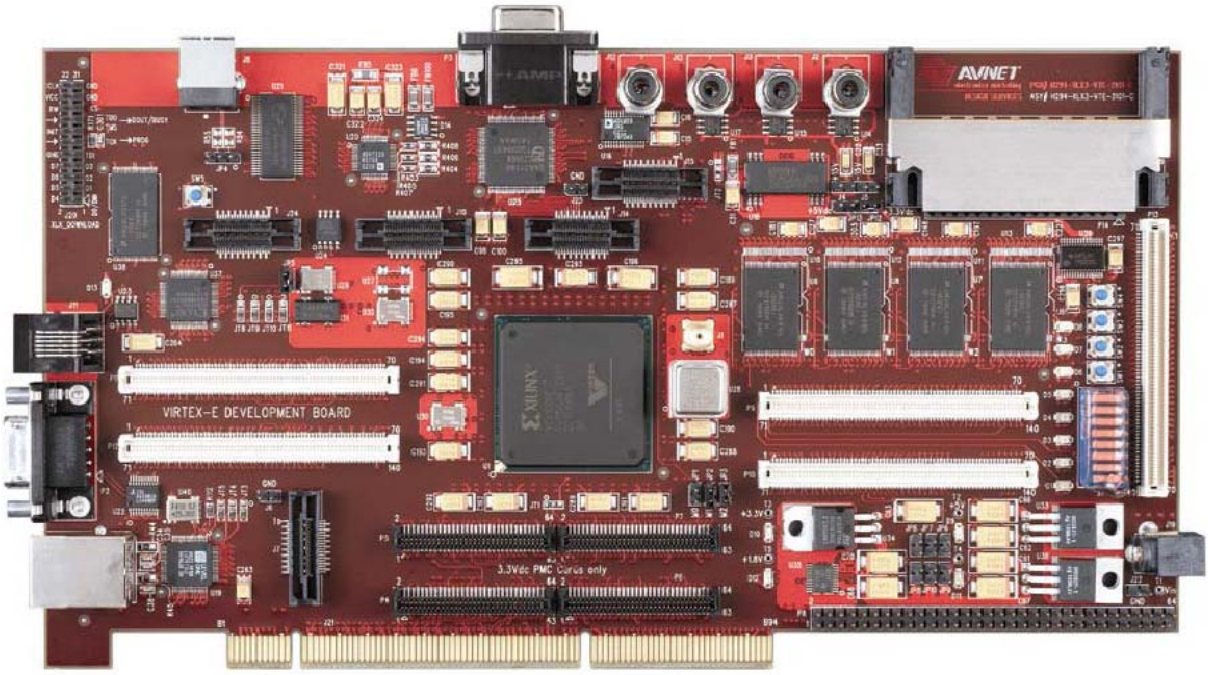


Abb. 2-1 AVNET Virtex–E 1000 Evaluation Board (Quelle: [D18])

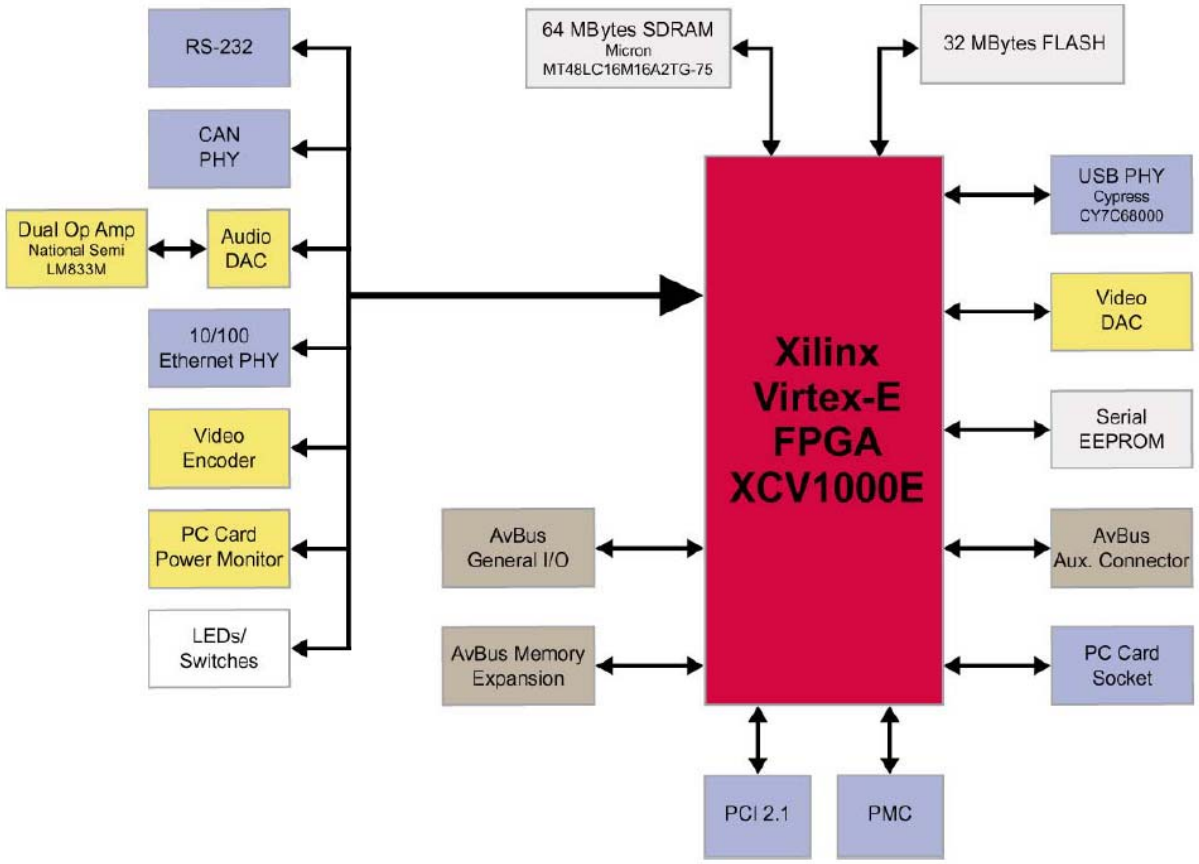


Abb. 2-2 AVNET Board Blockdiagramm (Quelle: [D18])

FPGA

Als FPGA steht ein XCV1000E -6 FP1156 der Firma Xilinx zu Verfügung. Es arbeitet mit einer Core-Spannung von 1,8 V und einer I/O-Treiber Spannung von 3,3 V. Die Eingänge sind **nicht** 5V tolerant. Das FPGA wird für Tests das gesamte LEON-Design enthalten und darüber Speicher und Schnittstellen ansteuern.

Die Konfiguration kann auf zwei Arten geschehen. Erstens über das JTAG–Interface bzw. durch Xilinx definierte Standards (z.B. über die Xilinx IMPACT Software) im Serial oder Parallel Mode. Dabei geht die Konfiguration bei Unterbrechung der Spannungsversorgung verloren. Zweitens kann über eine Kombination aus CPLD und Flash-Memory im SelectMap Modus konfiguriert werden. Dazu ist spezielle Software zum Beschreiben des Flashs und eine CPLD-Konfiguration notwendig, die vom Hersteller zu Verfügung gestellt werden. Bei dieser Art der Konfiguration wird das FPGA bei Anlegen der Spannungsversorgung automatisch konfiguriert.

Speicher

Zwei Speicherarten sind auf der Leiterplatte vorhanden. Beide sind in 64-Bit–Worten angeordnet und an denselben Speicherbus angeschlossen.

Als Flash-Memory stehen 4 Bausteine (AM29LV641 FLASH der Firma AMD) mit insgesamt 32 MB zu Verfügung. Da der Inhalt ohne Spannungsversorgung erhalten bleibt, eignet sich dieser Speichertyp für das LEON-Design als ROM bzw. Boot-ROM.

Der nicht permanente Speicher des ADS-XLX-VE-DEV besteht aus 4 SDRAM Bausteinen (MT48LC8M16A2TG der Firma Micron) mit einer Gesamtkapazität von 64MB. Dieser Speicher wird als Arbeitsspeicher für den LEON-Prozessor benutzt.

PCI-Bus

Die PCI-Schnittstelle unterstützt 32- oder 64-Bit Betrieb bei 33 MHz und 3,3V–Standard. Es wird allerdings nur der 32-Bit Betrieb getestet, da sowohl die verwendete AMBA-PCI-Bridge als auch das verwendete PC-Test-System nur mit 32 Bit arbeiten.

Es steht ein Standard Connector (d.h. das Board wird als PCI-Steckkarte verwendet) oder PMC-Connectoren (d.h. eine PCI Mezzanine Card wird eingesteckt) zu Verfügung. Die gleichzeitige Benutzung beider Verbindungen verbietet sich, da sonst mehr als ein PCI-Teilnehmer an einem PCI-Slot angeschlossen wäre und dies die PCI-Spezifikation nicht zulässt (vgl.[L5]).

RS232

Die serielle RS232 Schnittstelle ist inklusive Treiberbaustein (ADM3222 von Analog Devices) vorhanden und mit dem FPGA verbunden. Da das LEON-Design zwei allgemeine UARTs und ein Debug-Interface (auch UART) vorsieht, ist eine von Astrium selbst hergestellte Leiterplatte mit dem AVNET Board verbunden, die zwei RS232 Treiber und zwei DSUB9 Stecker beherbergt. Dadurch können drei serielle Schnittstellen gleichzeitig benutzt werden.

RJ45

Der Betrieb einer Ethernet-Schnittstelle ist optional.

Spannungsversorgung

Das Board wird entweder über den PCI-Bus (Board als PCI-Steckkarte) oder über einen externen Spannungsgenerator versorgt. Über Linearregler werden die benötigten Betriebsspannungen 3,3V und 1,8V erzeugt.

LEDs, Buttons, Switches

Es stehen 8 grüne LEDs zu Verfügung, die über das FPGA angesteuert werden können. Um Eingangssignale zu erzeugen stehen 8 DIP-Switches und 4 Push Buttons zu Verfügung.

2.1.2. PMC-Caddy

Als Trägerboard für das zu entwickelnde PMC-Board ist ein Produkt der Firma ESD GmbH vorgesehen. Das Board wird als PMC-Caddy bezeichnet und ist als VME-Karte ausgelegt, die Platz für zwei PMC-Karten bietet. Zwischen VME-Bus und PCI-Bus sitzt eine Bus-Bridge der Firma Tundra.

Da zwei PMC-Plätze vorhanden sind, kann das Board dafür genutzt werden, Kommunikation zwischen zwei PMC-Karten zu testen. Dies ist für das zu entwickelnde PMC-Board notwendig.

Abb. 2-3 und Abb. 2-4 zeigen das PMC-Caddy und das Blockschaltbild.

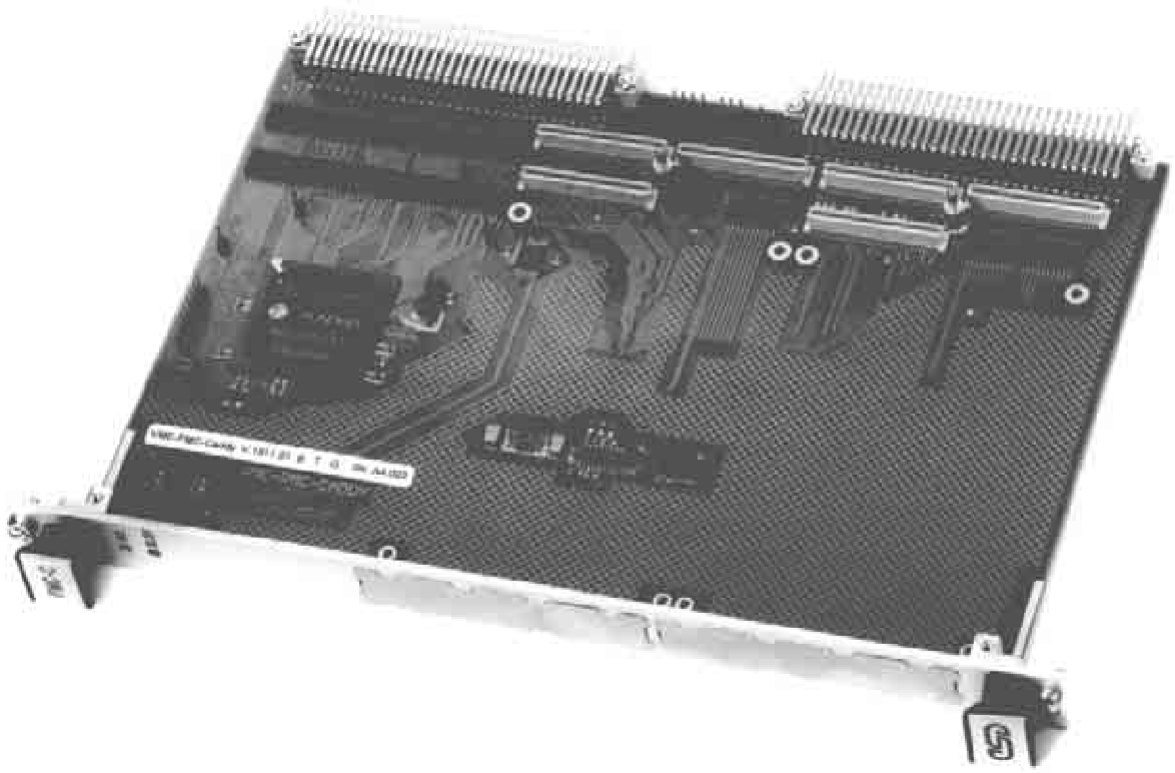


Abb. 2-3 PMC-Caddy (Quelle: [D16])

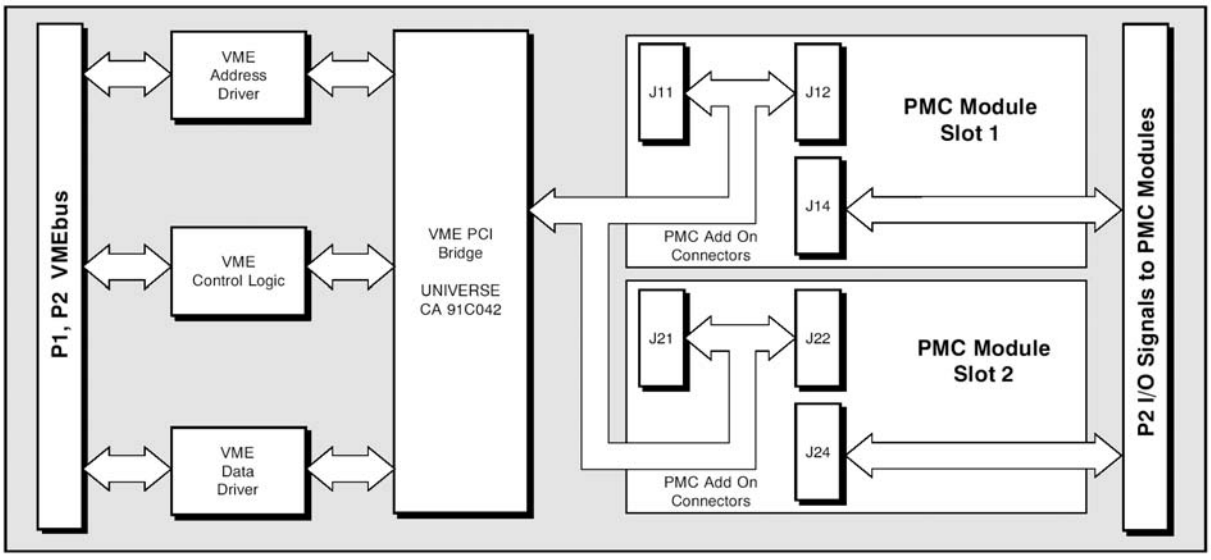


Abb. 2-4 Blockschaltbild des PMC-Caddys (Quelle: [D16])

2.1.3. PMC-Board

Das PMC-Board ist die Hardware, die im Rahmen der Diplomarbeit entwickelt und getestet werden soll. PMC steht für PCI Mezzanine Card. Mezzanine Cards sind Karten, die „Huckepack“ auf Trägerkarten sitzen. Diese Trägerkarten werden auch als Host-Karten bezeichnet. Die Schnittstelle zwischen Host und Mezzanine Card gibt PMC den Namen: Es

wird der PCI-Bus (Periphereal Component Interconnect) zur Kommunikation verwendet. PMC basiert auf der CMC-Spezifikation (Common Mezzanine Card), die konstruktive, mechanische, thermische und elektrische Vorgaben macht. Die PMC-Spezifikation beschreibt die Belegung der durch CMC beschriebenen Steckverbinder zwischen Host und Mezzanine Card. Einige Belegungen sind durch CMC schon vorgegeben. In den folgenden Kapiteln werden die beiden Spezifikationen kurz zusammengefasst, um sich im späteren Verlauf darauf zu beziehen.

2.1.3.1. CMC-Spezifikation

Die CMC-Spezifikation (Common Mezzanine Card) beschreibt konstruktive, mechanische und thermische Anforderungen, die unter dem Punkt *Mechanische Anforderungen* subsummiert sind. Elektrische und logische Vorgaben beziehen sich auf Bustyp-unabhängige Signale.

Mechanische Anforderungen

Die CMC-Spezifikation beschreibt zwei Mezzanine Card Größen: Single CMC und Double CMC. Die Diplomarbeit befasst sich mit einer Single CMC. Abb. 2-5 zeigt eine Single CMC (im folgenden CMC) mit ihren zwei Seiten. Seite 1 ist die Seite, die der Host Karte zugewandt ist und die Steckverbinder sowie I/O-Stecker beherbergt. Die Fläche des I/O-Bereiches wird als Front Panel, Front Bezel oder CMC Bezel bezeichnet. Seite 2 ist die dem Host abgewandte Seite. Abb. 2-6 illustriert, wie eine CMC-Karte auf einer Host Karte sitzt.

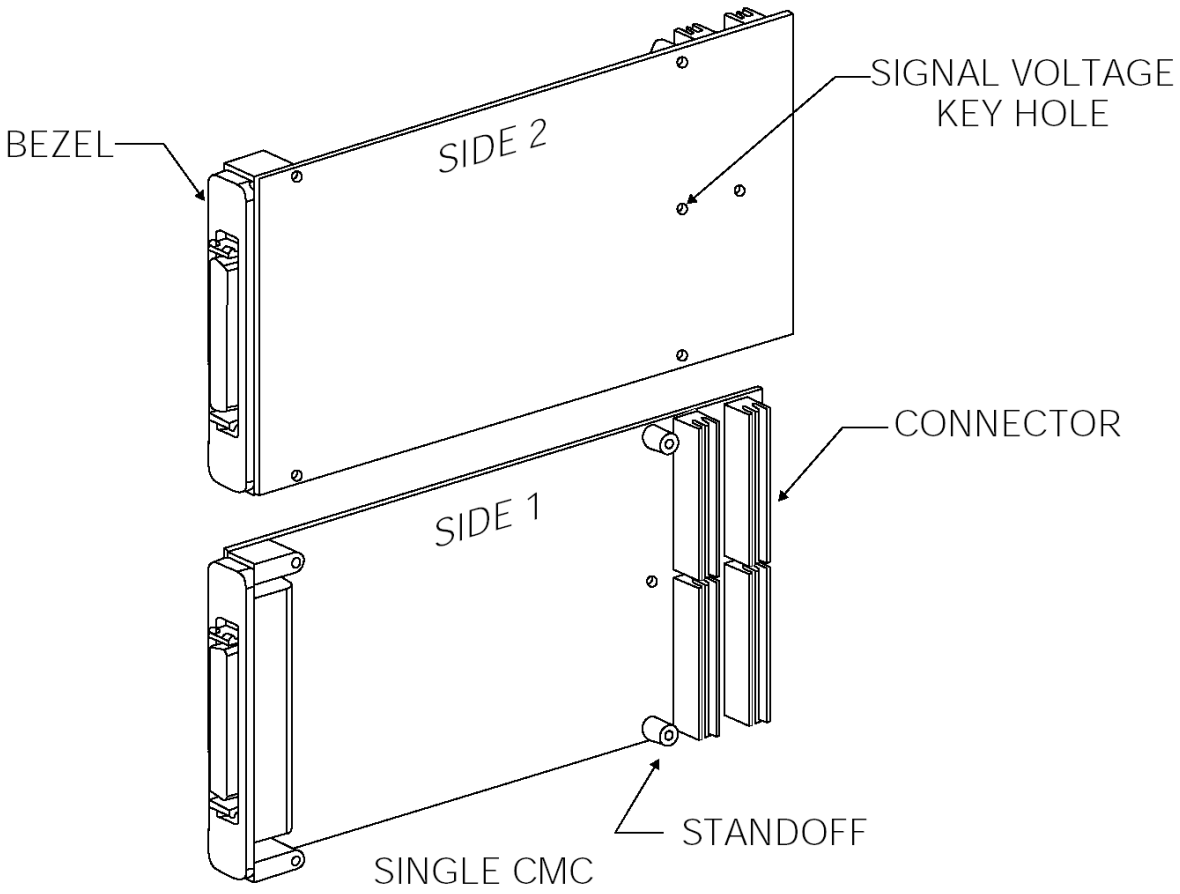


Abb. 2-5 Typische Single CMC (Quelle: [L7])

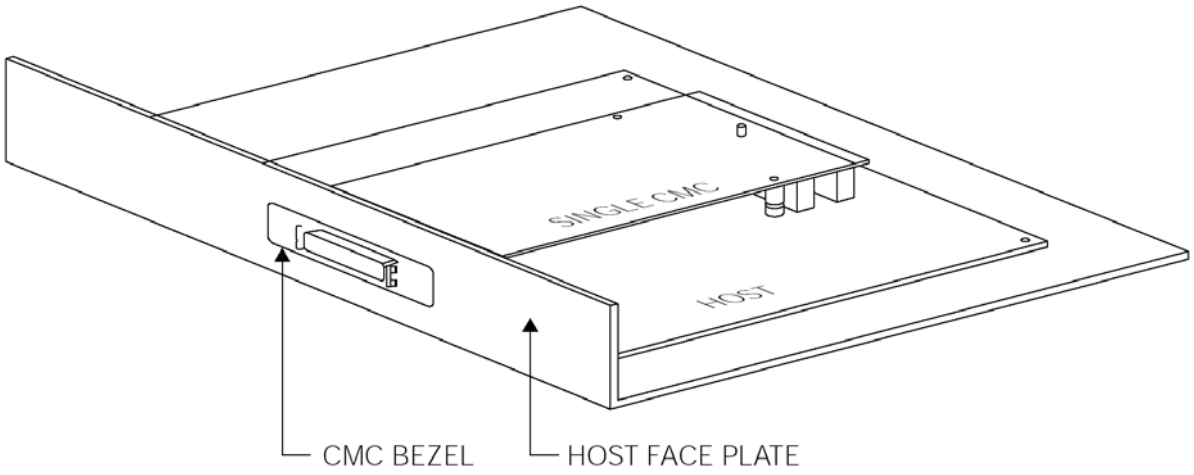


Abb. 2-6 CMC auf einer Host Karte (Quelle: [L7])

Die CMC-Spezifikation sieht zwei verschiedene Spannungspegel vor, mit denen der Bus zwischen Host und CMC arbeitet. Dies sind 3,3V und 5V. Durch zwei Löcher in der CMC kann der Host erkennen, welche Signalpegel unterstützt werden (s. Abb. 2-5). Sind beide Löcher vorhanden, werden von der CMC beide Signalpegel unterstützt.

Nicht alle vier Steckverbinder zwischen Host und CMC sind zwingend vorgeschrieben. Ein Steckverbinder ist für I/O-Signale vorgesehen, die nicht durch das Bussystem belegt sind. Die anderen drei sind für das Bussystem reserviert, mit Ausnahme der durch die CMC-Spezifikation festgelegten Pins (s.u.).

Eine Single CMC darf auf Seite 1 maximal 6.0 W an Leistung verbrauchen bzw. in Wärme umwandeln. Insgesamt darf sie bis maximal 7.5 W verbrauchen.

Bestimmte Anforderungen an EMV, Stoß- und Vibrationsfestigkeit sind in der Spezifikation aufgeführt. Diese werden hier nicht betrachtet.

Die CMC-Spezifikation beschreibt eine Einhüllende, innerhalb derer sich alle Komponenten der CMC befinden müssen. Dadurch wird gewährleistet, dass sich Host und CMC nicht gegenseitig stören. Diese Einhüllende wird in Abb. 2-7 gezeigt. Durch den Abstand des PCB vom Host, ist die maximale Bauteilhöhe auf Seite 1, Seite 2 und im I/O-Bereich definiert. Die Spezifikation sieht vor, dass durch unterschiedlich hohe Steckverbinder zwischen Host und CMC dieser Abstand im geringen Rahmen variiert werden kann, um größere Flexibilität zu gewährleisten. Die Einhüllende aus Abb. 2-7 bleibt dabei unverändert. Tabelle 2-1 gibt an, welche Bauteilhöhen benutzt werden können, bei verschiedenen Abständen zwischen Host und CMC. Der Standard ist mit einem Abstand von 10mm festgelegt.

Abstand Host-CMC (mm)	Seite 1 max. Bauteilhöhe (mm)	Seite 1 max. Bauteilhöhe im I/O-Bereich (mm)	Seite 2 max. Bauteilhöhe + PCB-Höhe (mm)
8,00	2,70	8,00	5,50
9,00	3,70	9,00	4,50
10,00	**4,70**	**10,00**	**3,50**
11,00	5,70	11,00	2,50

Tabelle 2-1 Beziehung zwischen Abstand Host-CMC und Bauteilhöhen

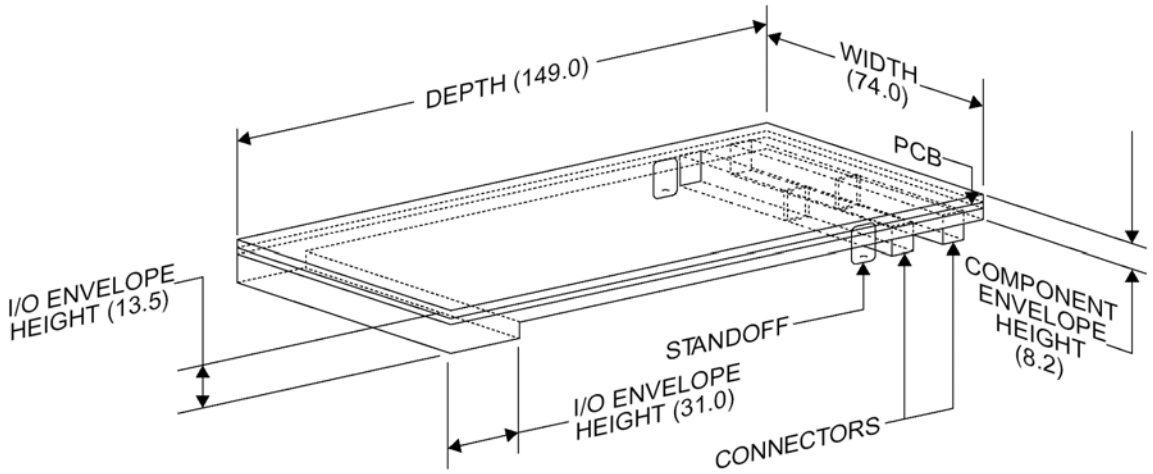


Abb. 2-7 Einhüllende einer Single CMC (Quelle: [L7])

Elektrische und logische Vorgaben

Die elektrischen Vorgaben betreffen vor allem die Belegung der Connectoren zwischen Host und CMC. Sie legen Versorgungsspannungen und BUSMODE#-Signale für die Stecker 1-3 fest. Der Stecker 4 wird ausschließlich mit I/O-Leitungen belegt. Die genaue Belegung für **PMC** ist in Kapitel 2.1.3.2 in Tabelle 2-3 und Tabelle 2-4 zu sehen. Sie beinhaltet die durch die CMC-Spezifikation vorgeschriebenen Pins.

Die BUSMODE#-Signale werden von dem Host dazu benutzt, die Anwesenheit einer CMC und das von ihr benutzte Busprotokoll festzustellen. Es gibt 4 BUSMODE#-Signale. BUSMODE#(2..4) werden vom Host getrieben und sind mit allen vorhandenen CMC-Steckplätzen verbunden. Für jeden CMC-Steckplatz steht eine BUSMODE#(1) Signalleitung zu Verfügung, die alle zur Hostlogik führen, und nur von den eingesteckten Mezzanine Cards getrieben werden.
In Abhängigkeit der Signale BUSMODE#(2..4) antwortet jede eingesteckte CMC auf BUSMODE#(1).
Die Signale auf BUSMODE#(2..4) codieren eine Betriebsart, die durch Tabelle 2-2 beschrieben wird. Wenn die CMC diese Betriebsart unterstützt zieht sie BUSMODE#(1) auf Low und schaltet intern auf das signalisierte Protokoll um. Wie aus Tabelle 2-2 zu sehen, wird bisher nur ein Busprotokoll benutzt. Reservierte Kodierungen sind für die Zukunft vorgesehen.

BUSMODE4#	BUSMODE3#	BUSMODE2#	Beschreibung
L	L	L	Test, ob Karte vorhanden. Wenn CMC vorhanden, BUSMODE1# →Low. Kein Busprotokoll benutzen.
L	L	H	BUSMODE1#→Low, wenn Karte vorhanden und PCI-fähig. PCI Protokoll benutzen.
L	H	L	reserviert
L	H	H	reserviert
H	L	L	reserviert
H	L	H	reserviert
H	H	L	reserviert
H	H	H	kein Host vorhanden

Tabelle 2-2 BUSMODE# Kodierung

2.1.3.2. PMC-Spezifikation

Die PMC-Spezifikation ist ein Zusatz zur CMC-Spezifikation, der beschreibt, wie die Connectoren zwischen Host und Mezzanine Card (PMC-Steckverbinder) mit busspezifischen Signalen belegt sind. Stecker 4 ist optional und vollständig durch die CMC-Spezifikation beschrieben.

Stecker 1 und 2 werden für den 32-Bit PCI-Bus verwendet. Zusätzliche Signale für 64-Bit PCI-Betrieb liegen an Stecker 3. Dieser ist also für 32-Bit PCI überflüssig und muss nicht auf einer PMC vorhanden sein. Tabelle 2-3 und Tabelle 2-4 stellen die vorgeschriebene Pinbelegung der PMC-Steckverbinder dar.

Stecker 1 (Pn1): 32-Bit PCI				Stecker 2 (Pn2): 32-Bit PCI			
Pin	**Signal**	**Signal**	**Pin**	**Pin**	**Signal**	**Signal**	**Pin**
1	TCK	-12V	2	1	+12V	TRST#	2
3	Ground	INTA#	4	3	TMS	TDO	4
5	INTB#	INTC#	6	5	TDI	Ground	6
7	BUSMODE1#	+5V	8	7	Ground	PCI-RSVD*	8
9	INTD#	PCI-RSVD*	10	9	PCI-RSVD*	PCI-RSVD*	10
11	Ground	3.3Vaux	12	11	BUSMODE2#	+3.3V	12
13	CLK	Ground	14	13	RST#	BUSMODE3#	14
15	Ground	GNT#	16	15	3.3V	BUSMODE4#	16
17	REQ#	+5V	18	17	PME#	Ground	18
19	V (I/O)	AD[31]	20	19	AD[30]	AD[29]	20
21	AD[28]	AD[27]	22	21	Ground	AD[26]	22
23	AD[25]	Ground	24	23	AD[24]	+3.3V	24
25	Ground	C/BE[3]#	26	25	IDSEL	AD[23]	26
27	AD[22]	AD[21]	28	27	+3.3V	AD[20]	28
29	AD[19]	+5V	30	29	AD[18]	Ground	30
31	V (I/O)	AD[17]	32	31	AD[16]	C/BE[2]#	32
33	FRAME#	Ground	34	33	Ground	PMC-RSVD	34
35	Ground	IRDY#	36	35	TRDY#	+3.3V	36
37	DEVSEL#	+5V	38	37	Ground	STOP#	38
39	Ground	LOCK#	40	39	PERR#	Ground	40
41	PCI-RSVD*	PCI-RSVD*	42	41	+3.3V	SERR#	42
43	PAR	Ground	44	43	C/BE[1]#	Ground	44
45	V (I/O)	AD[15]	46	45	AD[14]	AD[13]	46
47	AD[12]	AD[11]	48	47	M66EN	AD[10]	48
49	AD[09]	+5V	50	49	AD[08]	+3.3V	50
51	Ground	C/BE[0]#	52	51	AD[07]	PMC-RSVD	52
53	AD[06]	AD[05]	54	53	+3.3V	PMC-RSVD	54
55	AD[04]	Ground	56	55	PMC-RSVD	Ground	56
57	V (I/O)	AD[03]	58	57	PMC-RSVD	PMC-RSVD	58
59	AD[02]	AD[01]	60	59	Ground	PMC-RSVD	60
61	AD[00]	+5V	62	61	ACK64#	+3.3V	62
63	Ground	REQ64#	64	63	Ground	PMC-RSVD	64

Tabelle 2-3 PMC-Steckverbinder Pn1 / Pn2

Stecker 3 (Pn3): 64-Bit PCI				Stecker 4 (Pn4): User Defined I/O			
Pin	**Signal**	**Signal**	**Pin**	**Pin**	**Signal**	**Signal**	**Pin**
1	PCI-RSVD	Ground	2	1	I/O	I/O	2
3	Ground	C/BE[7]#	4	3	I/O	I/O	4
5	C/BE[6]#	C/BE[5]#	6	5	I/O	I/O	6
7	C/BE[4]#	Ground	8	7	I/O	I/O	8
9	V (I/O)	PAR64	10	9	I/O	I/O	10
11	AD[63]	AD[62]	12	11	I/O	I/O	12
13	AD[61]	Ground	14	13	I/O	I/O	14
15	Ground	AD[60]	16	15	I/O	I/O	16
17	AD[59]	AD[58]	18	17	I/O	I/O	18
19	AD[57]	Ground	20	19	I/O	I/O	20
21	V (I/O)	AD[56]	22	21	I/O	I/O	22
23	AD[55]	AD[54]	24	23	I/O	I/O	24
25	AD[53]	Ground	26	25	I/O	I/O	26
27	Ground	AD[52]	28	27	I/O	I/O	28
29	AD[51]	AD[50]	30	29	I/O	I/O	30
31	AD[49]	Ground	32	31	I/O	I/O	32
33	Ground	AD[48]	34	33	I/O	I/O	34
35	AD[47]	AD[46]	36	35	I/O	I/O	36
37	AD[45]	Ground	38	37	I/O	I/O	38
39	V (I/O)	AD[44]	40	39	I/O	I/O	40
41	AD[43]	AD[42]	42	41	I/O	I/O	42
43	AD[41]	Ground	44	43	I/O	I/O	44
45	Ground	AD[40]	46	45	I/O	I/O	46
47	AD[39]	AD[38]	48	47	I/O	I/O	48
49	AD[37]	Ground	50	49	I/O	I/O	50
51	Ground	AD[36]	52	51	I/O	I/O	52
53	AD[35]	AD[34]	54	53	I/O	I/O	54
55	AD[33]	Ground	56	55	I/O	I/O	56
57	V (I/O)	AD[32]	58	58	I/O	I/O	58
59	PCI-RSVD	PCI-RSVD	60	59	I/O	I/O	60
61	PCI-RSVD	Ground	62	61	I/O	I/O	62
63	Ground	PCI-RSVD	64	63	I/O	I/O	64

Tabelle 2-4 PMC-Steckverbinder Pn3 / Pn4

2.2. VHDL-Cores

2.2.1. LEON

Der LEON VHDL-Core wurde ursprünglich von der ESA entwickelt und wird mittlerweile von der Firma Gaisler Research als Open-Core zu Verfügung gestellt. Die in dieser Diplomarbeit benutzte Version 10 ist mittlerweile veraltet. Der Core beschreibt einen 32-Bit Prozessor nach dem SPARC V8 – Standard. Zusätzlich zu dem Prozessor stellt das VHDL-Modell noch weitere Funktionalität zu Verfügung, die als SoC (System on Chip) in einem FPGA Platz findet. So werden getrennte Cache-Speicher für Daten und Befehle, Hardware Multiplizierer und Dividierer, Interrupt-Controller, zwei 24-Bit-Timer, zwei UARTs, Power-Down Funktion, Watchdog, 16-Bit I/O-Port, Speicher-Controller und eine Debug-Einheit mit dem LEON-Core mitgeliefert. Die Integer Unit des LEON stellt ein Interface zu einer *Floating Point Unit* (FPU) und einem Coprozessor (CP) zu Verfügung. Der erhältliche Code enthält jedoch keine Komponente, die diese Funktionalität beherbergt.

Als interner Bus wird ein AMBA-AHB-Bus (im folgenden AHB-Bus) verwendet. Konfigurationen der einzelnen Komponenten werden über den AMBA-APB-Bus (im folgenden APB-Bus) durchgeführt. Dieser APB-Bus ist über eine AHB-APB-Bridge an den AHB-Bus angekoppelt. Über diese beiden Bussysteme können auch beliebige weitere Komponenten an das System angebunden und implementiert werden (dies soll z.B. mit der AMBA-PCI-Bridge geschehen).

Abb. 2-8 zeigt das Blockdiagramm des LEON. PROM, I/O, SRAM, SDRAM sind externe Komponenten und werden nicht durch den Open-Core abgedeckt. Für FPU, CP, PCI, USER I/O sind Interfaces vorgesehen, die Funktionalität der Komponenten muss jedoch durch eigene VHDL-Module beschrieben oder von Drittanbietern erworben werden. Im Folgenden werden die Aufgaben und Funktionsweisen der einzelnen Teilkomponenten beschrieben.

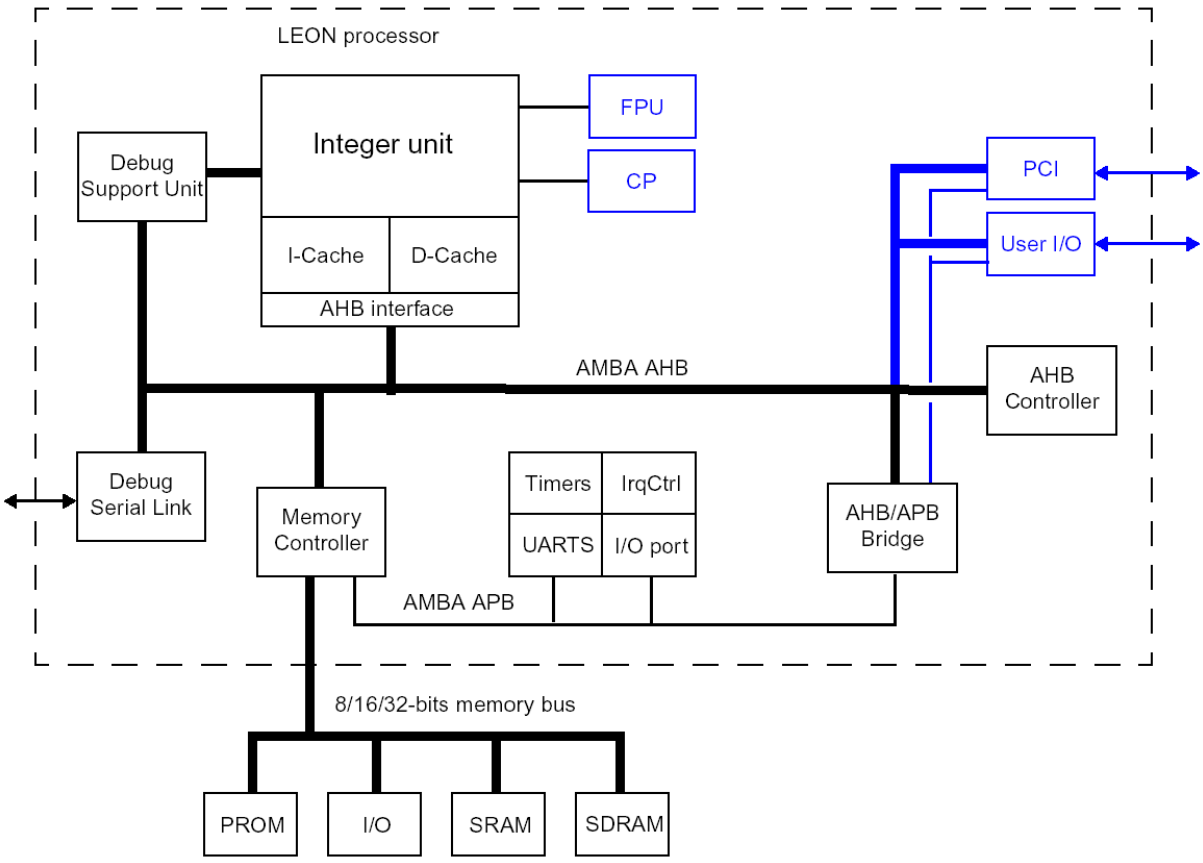


Abb. 2-8 Blockdiagramm des LEON (Quelle: [L10])

2.2.1.1. Integer Unit (IU)

Die IU des LEON ist nach dem SPARC V8 Standard entwickelt. Sie enthält den Registersatz des Prozessors und hat die Aufgabe, Befehle abzuarbeiten und Integer-Rechenoperationen durchzuführen. Dazu hat sie eine 5-stufige Pipeline-Verarbeitung mit den Arbeitsschritten Instruction Fetch (FE), Decode (DE), Execute (EX), Memory (ME) und Write (WR).

FE: In dieser Stufe wird ein Befehl aus dem Befehls-Cache geholt. Sollte dieser Cache nicht aktiviert sein (konfigurierbar), wird der Befehl aus dem Speicher geholt.

DE: Der Befehl wird dekodiert, Operanden werden gelesen.

EX: Die mathematischen Operationen (Addieren, logische Operationen, Schieben) werden hier durchgeführt.

ME: Der Daten-Cache wird in dieser Stufe angesteuert. Es werden sowohl Lese- als auch Schreibbefehle durchgeführt.

WR: Das Ergebnis der EX-Stufe wird in den internen Registersatz zurückgeschrieben.

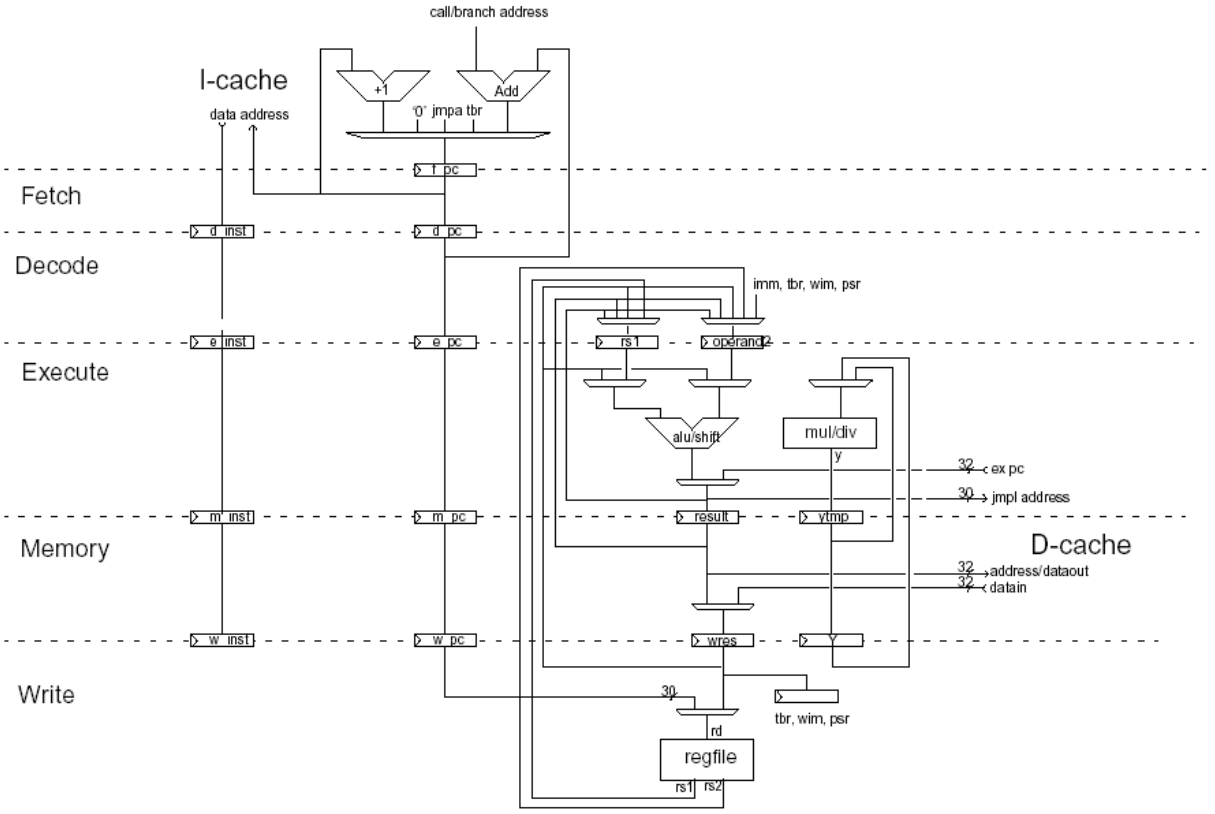


Abb. 2-9 LEON IU Blockdiagramm (Quelle: [L10])

Wie aus Abb. 2-9 zu erkennen ist, hat der LEON zwei unterschiedliche Arten von Cache implementiert. Einer ist für Befehle zuständig, der andere für Daten.

Der implementierte Hardware-Multiplizierer ist konfigurierbar bezüglich seiner Größe. Dies bedeutet, dass Multiplikationen großer Zahlen mit einem erhöhten Platzaufwand beschleunigt werden können, da sie weniger Iterationen bis zum Ergebnis brauchen.

Register

Die Register von SPARC-Prozessoren werden in Allgemein- („Arbeits-„) und Kontroll/Status-Register aufgeteilt. Arbeitsregister werden als Zwischenspeicher zur Abarbeitung von Programmen benutzt. Kontroll/Status-Register haben die Aufgabe, einerseits den Status, in dem sich der Prozessor befindet, zu speichern, um mit diesen Informationen weiterzuarbeiten, andererseits die Hardware zu kontrollieren. Beide Aufgabentypen werden zum Teil nicht streng in Register getrennt, sondern sind in einem Register vereint.

Allgemein-Register / Register – Fenster

Allgemeine Register setzen sich aus 8 globalen Registern und einer implementierungsabhängigen Anzahl von Register-Sätzen zusammen. Diese Registersätze bestehen aus 16 Registern. Der LEON kann zwischen 2 und 32 solcher Register-Sätze implementieren. Die Register-Sätze bestehen jeweils aus 8 sogenannten *in*-Registern und 8 sog. *local*-Registern.

Jedes Programm, das auf dem Prozessor abgearbeitet wird, hat Zugriff auf die 8 globalen Register und ein Fenster von weiteren 24 Registern. Dieses Register-Fenster besteht aus

einem Register-Satz (also 8 in-Register und 8 local-Register) und zusätzlich den 8 in-Registern des „angrenzenden“ Register-Satzes. Diese werden out-Register genannt. Ein aktuell laufender Prozess hat also immer nur Zugriff auf maximal 32 Register. Die Registernummer und relative Adresse (bzw. der Assembler-Name) ist dabei für alle Fenster identisch. Deshalb können Prozesse in jedem Fenster abgearbeitet werden, ohne dass der Programmcode angepasst werden muss.

Name	Assembler-Bezeichnung	Register-Adresse
in[0] – in[7]	%i0 - %i7	r[24] – r[31]
local[0] - local[7]	%l0 - %l7	r[16] – r[23]
out[0] – out[7]	%o0 - %o7	r[8] – r[15]
global[0] – global[7]	%g0 - %g7	r[0] – r[7]

Tabelle 2-5 Register-Adressierung und Benennung innerhalb eines Register-Fensters

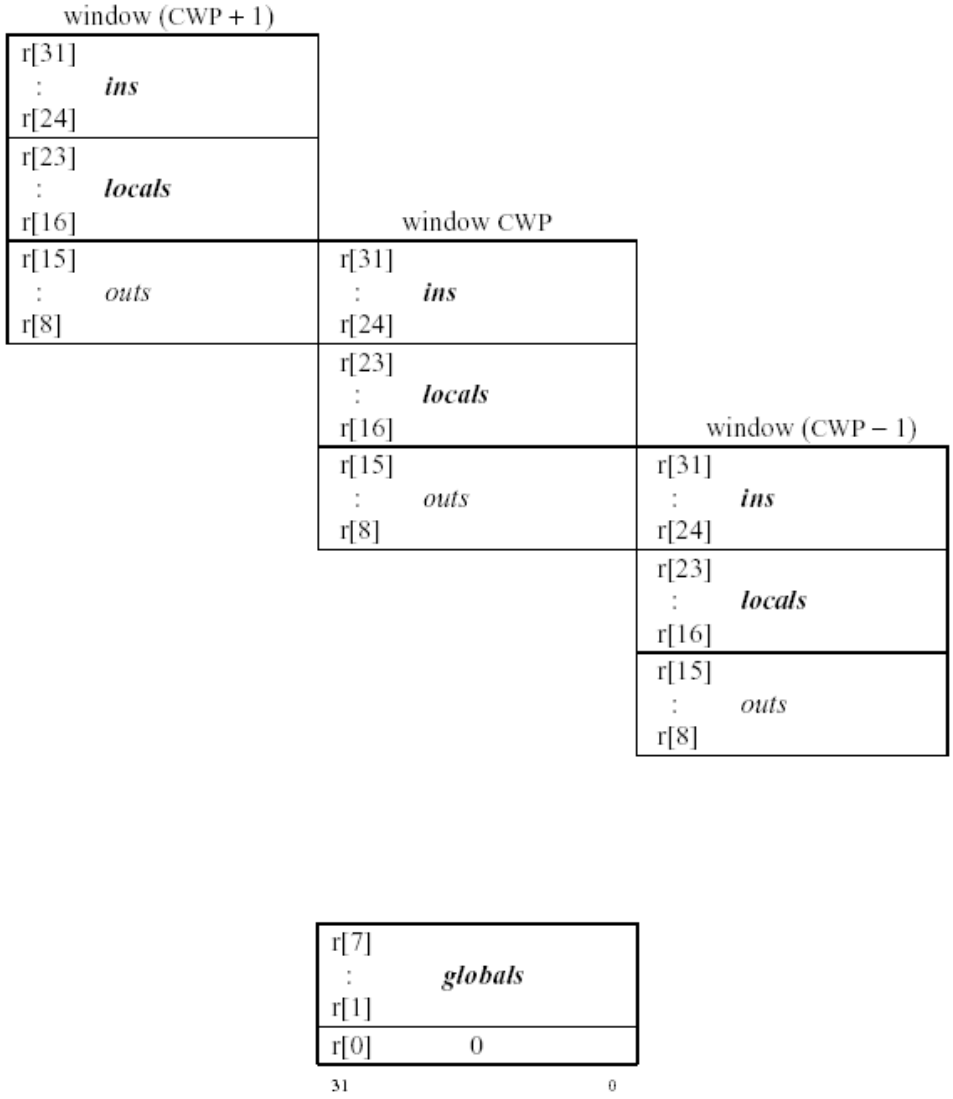


Abb. 2-10 3 überlappende Register-Fenster und die globalen Register (Quelle: [L14])

Abb. 2-10 zeigt 3 verschiedene Register-Fenster und ihre Beziehungen zueinander. Die globalen Register sind von allen Prozessen aus adressierbar und besonders dafür geeignet,

Werte zu speichern, die von vielen Prozessen verwendet werden. Dies betrifft z.B. die Anfangsadresse des Speicherbereiches.

Die in-Register des aktuellen Prozesses können mit Daten des aufrufenden Prozesses belegt sein. Sie sind also auch von dem aufrufenden Prozess als out-Register adressierbar.

Die local-Register sind ausschließlich vom aktuellen Prozess verwendbar.

Die out-Register werden zur Übergabe von Parametern beim Aufruf von Subroutinen benutzt. Liefert die Subroutine Rückgabewerte, werden diese sich auch wieder in den out-Registern auffinden lassen. Die out-Register des aktuellen Prozesses sind als in-Register vom nächsten Prozess aus adressierbar.

Beispiel:
Ein Programm arbeitet mit den ihm zu Verfügung stehenden Registern. Wird eine Subroutine aufgerufen, die Eingangsparameter benötigt, schreibt das aufrufende Programm diese vorher in seine out-Register. Das Register-Fenster wird beim Aufruf der Routine gewechselt, d.h. die Routine hat – wie alle Prozesse – ihr eigenes 24-Register-Fenster. Die vom aufrufenden Programm übergebenen Parameter befinden sich jetzt (relativ gesehen) in den in-Registern (daher die Namensgebung). Die Routine arbeitet mit den übergebenen Parametern z.B. dann in den lokalen Registern. Wird eine weitere Routine von hier aufgerufen, werden die Parameter wieder in die Output-Register geschrieben, usw. Hat die Routine jedoch einen Rückgabewert, wird dieser in die Input-Register geschrieben. Beim Beenden der Routine wird wieder in die aufrufende Instanz zurückgesprungen und damit auch wieder das Fenster gewechselt. Der eventuell vorhandene Rückgabewert steht nun in den Output-Registern und kann von dem (in diesem Beispiel) ursprünglich aufrufenden Programm lokal weiterverarbeitet werden.

Das aktuelle Register-Fenster wird durch den *Current Window Pointer* (CWP) bestimmt. Der CWP ist ein 5-Bit Feld, das als Zähler benutzt wird und innerhalb des *Processor State Register* (PSR) zu finden ist. Der CWP wird inkrementiert, wenn ein RESTORE oder ein RETT Befehl ausgeführt wird. Dies bedeutet, dass ein Programm oder Unterprogramm beendet wird. Dekrementiert wird der CWP, wenn ein SAVE Befehl ausgeführt wird oder sich ein Trap ereignet (s. Traps). Ersteres geschieht bei einem (Unter-) Programmaufruf, letzteres z.B. bei einem Interrupt (Interruptbehandlung geschieht durch Software). Welche Register-Fenster schon benutzt werden, ist im *Window invalid Mask Register* (WIM) gespeichert.

Anmerkung: Subroutinen und (Unter-)Programme können auch durch andere Befehle aufgerufen werden. Dadurch können Aufrufe auch ohne Wechsel des Register-Fensters gemacht werden.

Der CWP ist nach SPARC-Standard als modulo NWINDOWS Zähler implementiert, wobei NWINDOWS die Anzahl der Register-Fenster ist. Dies bedeutet, dass das Fenster mit der höchsten Nummer (NWINDOWS-1) mit dem Fenster 0 (Null) in der oben beschriebenen Weise überlappt. Genauer: Die out-Register von Fenster 0 sind identisch mit den in-Registern von Fenster NWINDOWS-1. Abb. 2-11 verdeutlicht, dass alle Fenster in einem Ring angeordnet sind. Eine Konsequenz von fortlaufenden SAVE Befehlen (oder Traps) wäre, dass nach einer bestimmten Anzahl derselben ein Überlaufen stattfinden würde. Beispielsweise würde bei NWINDOWS = 7 und Start bei CWP = 7 beim 8. SAVE Befehl (ohne dazwischenliegendes RESTORE o.ä.) wieder das Fenster Nr. 7 selektiert.

Überlaufen (*Window Overflow*) wie dieses wird von der Hardware mithilfe des WIM festgestellt (WIM[7]=1) und muss durch Software behandelt werden.

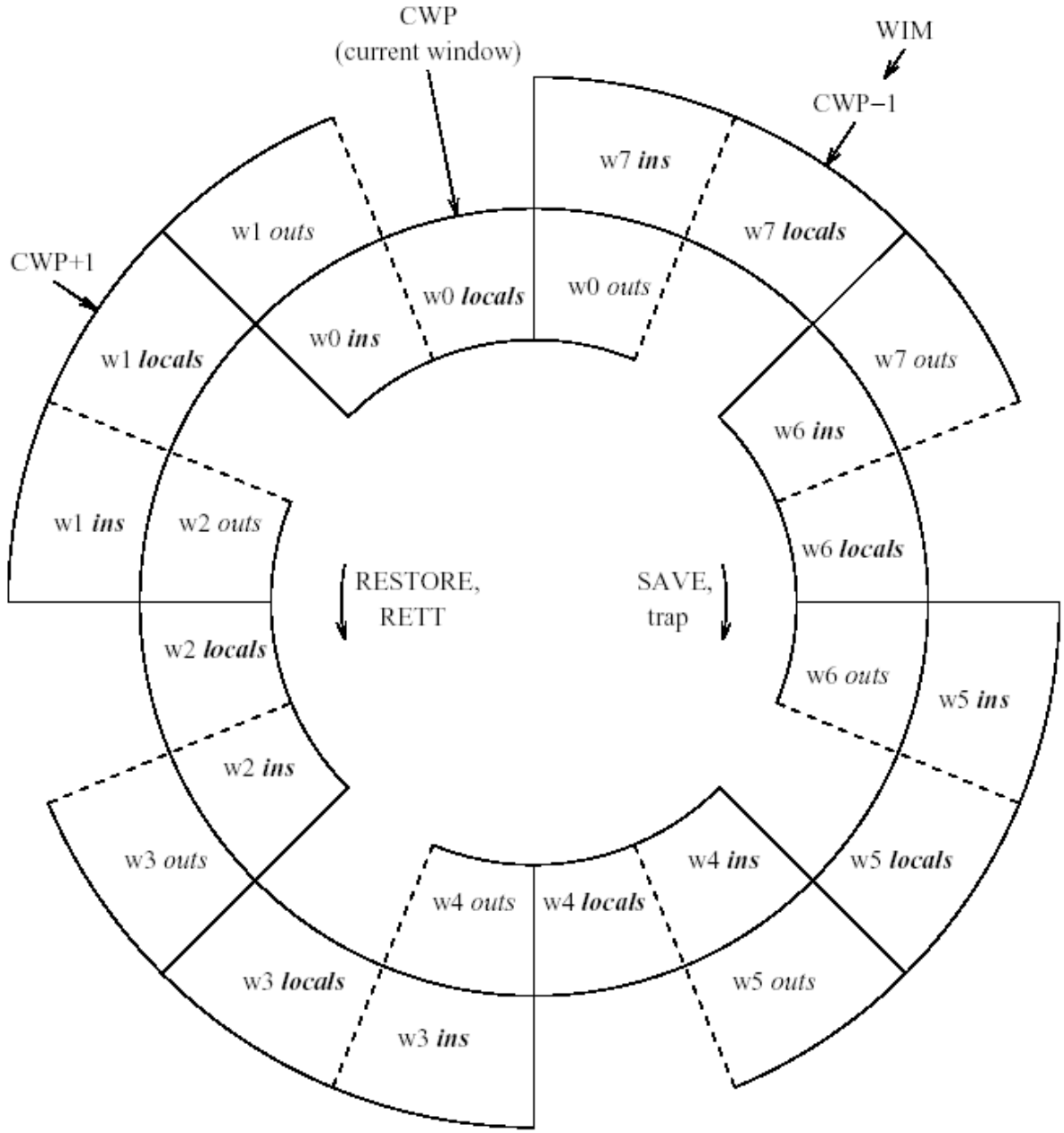


Abb. 2-11 Allg. Registersatz mit 8 Fenstern (ohne globale Register) (Quelle: [L14])

Kontroll/Status-Register

Processor State Register (PSR)

Das PSR enthält verschiedene Felder zur Kontrolle des Prozessors und zum Speichern von Statusinformationen. Zusätzlich sind hier Daten über die implementierte Architektur und die Versionsnummer festverdrahtet abgelegt.

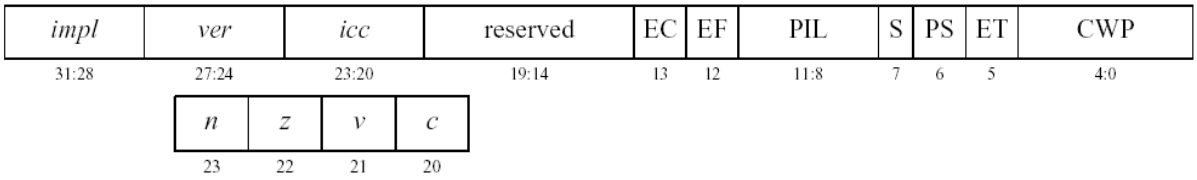


Abb. 2-12 Processor State Register (PSR) (Quelle: [L10])

Integer condition codes (icc): Die Bits werden durch die ALU modifiziert.
- n-Bit(negative): Zeigt ein negatives Ergebnis an (1=negativ, 0=nicht negativ)
- z-Bit(zero): Zeigt ein Ergebnis von Null an(1=Null, 0=nicht Null)
- v-Bit(overflow): Zeigt an, ob das Ergebnis außerhalb des Wertebereiches eines 32-Bit 2-Komplement ist (1=Überlauf, 0=im Wertebereich)
- c-Bit(carry): Zeigt einen Übertrag an (1=Übertrag, 0=kein Übertrag)

Enable Coprocessor(EC): Erlaubt die Benutzung des Coprozessors (1=enabled, 0=disabled)

Enable Floating Point(EF): Erlaubt die Benutzung der FPU (1=enabled, 0=disabled)

Processor Interrupt Level(PIL): Legt den Level fest, **über** dem auftretende Interrupts behandelt werden. Je höher der Level eines Interruptes, desto höher ist seine Priorität. (Siehe Unterpunkt *SPARC Trap Model.*)

Supervisor(S): Der Prozessor kann entweder in User- oder Supervisor-Mode betrieben werden(1=supervisor mode , 0=user mode).

Previous Supervisor(PS): Enthält den Wert von S beim letzten Trap.

Enable Traps(ET): Gibt an, ob Traps und Interrupts behandelt werden (1=enabled, 0=disabled).

Current Window Pointer(CWP): Ein modulo NWINDOWS Zähler, der festlegt, welches Register-Fenster aktiv ist. NWINDOWS entspricht der Anzahl von Register-Fenstern, die implementiert sind.

Window Invalid Mask (WIM)

Für jedes implementierte Register-Fenster ist ein veränderbares Bit im WIM implementiert. Andere Bits sind festverdrahtet auf Null gelegt. Eine 1 signalisiert, dass das zugehörige Register-Fenster schon benutzt wird. Das Register wird durch Systemsoftware kontrolliert. Wird ein SAVE, RESTORE oder RETT Befehl ausgeführt, überprüft die Hardware anhand des CWP, ob der Befehl den CWP auf ein schon benutztes Register-Fenster zeigen lässt. Ist dies der Fall, d.h. ist das WIM an der entsprechenden Stelle 1, wird ein *Window Overflow* Trap ausgelöst.

Trap Base Register (TBR)

Die Trap Table (siehe SPARC Trap Modell) kann variabel im Adressraum abgelegt werden. Das TBR enthält die Anfangsadresse dieses Adressraumes, die Trap Base Address (TBA). Der Trap Type (TT) Eintrag wird durch die Hardware erzeugt, wenn ein Trap auftritt. Die

unteren Bits sind Null. Der Inhalt dieses Registers stellt die absolute Adresse dar, zu der gesprungen wird, wenn sich ein Trap ereignet und ET=1 ist.

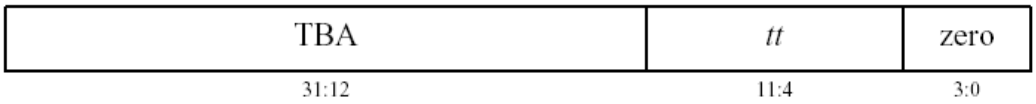


Abb. 2-13 Trap Base Register (TBR) (Quelle: [L10])

Program Counters (PC, nPC)

Der PC enthält die Adresse des Befehls, der gerade ausgeführt wird. Der nPC (next Program Counter) enthält die Adresse des nächsten auszuführenden Befehls. Nach einem Reset oder nach dem Einschalten wird der PC auf 0x0 und der nPC auf 0x4 initialisiert.

Multiply/Divide Register (Y)

Dieses Register enthält die höherwertigen 32-Bit einer Multiplikation oder Division mit doppelter Genauigkeit.

Ancillary State Register (ASR)

Bis zu 31 Register werden durch die SPARC Spezifikation unterstützt. Davon sind die Register 1 – 15 für die Zukunft reserviert. Die Register 16 – 31 können je nach Implementierung für Timer, Diagnose o.ä. verwendet werden.

Der LEON benutzt die Register 18 und 24 – 31.
ASR 18 wird als Hilfsregister für die *Multiply and Accumulate* (MAC)-Befehle verwendet. Es gibt zwei verschiedene MAC-Befehle: *unsigned* (UMAC) und *signed* (SMAC).

ASR 24 – 31 werden für sog. *Watchpoints* benutzt. Das sind vom Benutzer über Software festlegbare Adressen, an denen die Abarbeitung eines Programms gestoppt wird, um z.B. eine Überprüfung der IU- oder LEON-Register oder des Speicherinhaltes zu ermöglichen. Dabei kann unterschieden werden, ob ein Befehl oder ein Datenwort von der festgelegten Adresse gelesen (bzw. geschrieben) werden soll.

Ein Watchpoint besteht aus einem Register-Paar (24/25, 26/27, 28/29, 30/31). Davon enthält ein Register die Adresse (WADDR Feld), das andere eine Maske(WMASK Feld), die jedes Bit einzeln zum Vergleich zulässt oder nicht (1=vergleichen, 0=nicht vergleichen). Es kann nicht auf einzelne Bytes innerhalb eines Wortes geprüft werden, d.h. es werden nur Anfangsadressen von 32-Bit-Worten verarbeitet. Deshalb sind in beiden Registern die unteren beiden Bits für andere Zwecke vorgesehen. Über die Felder *instruction fetch*(IF), *data load*(DL) und *data store*(DF) kann festgelegt werden, welche Operation bei Zugriff auf die Adresse in WADDR durchgeführt werden muss, um einen Treffer auszulösen. Sind alle Felder deaktiviert, entspricht dies einem Deaktivieren der Watchpoint-Funktion. Ein Treffer wird über die Generierung eines Traps der IU angezeigt. Die Anzahl der im LEON implementierten Watchpoints ist konfigurierbar (bis 4). Abb. 2-14 zeigt die Belegung der Register.

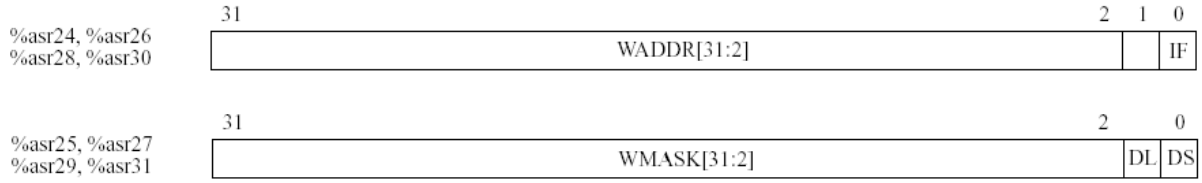


Abb. 2-14 Watchpoint Register (Quelle: [L10])

SPARC Trap Modell

Die SPARC V8 Spezifikation beschreibt eine Methode, um mit Fehlern, Ausnahmen bzw. unvorhersehbaren Ereignissen umzugehen. Solche Ereignisse können darin bestehen, dass Fehler während Lese- oder Schreibbefehlen auftreten, dass ein eventueller Coprozessor oder eine eventuell vorhandene FPU eine Ausnahmebedingung anzeigt oder dass Interrupts ausgelöst werden. Die SPARC V8 Spezifikation bezeichnet das als Trap, was übersetzt Falle heißt.
Ein Trap ist ein unerwarteter Prozessaufruf. Ob ein Trap behandelt wird, hängt vom Eintrag im ET-Feld des PSR ab. Bei Behandlung wird das Register-Fenster gewechselt. Um später wieder zu dem Programmpunkt zurückzukehren, werden PC und nPC in den lokalen Registern des Trap-Fensters abgespeichert. Ein Trap kann durch die Abarbeitung von Befehlen oder durch externe Interrupt Anforderungen entstehen. Es werden drei Arten von Traps unterschieden:

- *Precise trap*: Ein precise Trap wird durch einen Befehl (bzw. während dessen Abarbeitung) ausgelöst, ohne dass sich vom Programm aus sichtbare Zustände schon geändert haben. Der Befehl vor dem Trap-verursachenden muss schon abgearbeitet sein und der darauffolgende wird nicht ausgeführt.
- *Deferred trap*: Dieser wird wie ein precise Trap durch einen Befehl ausgelöst. Im Gegensatz zu diesem ereignet sich der deferred Trap jedoch nachdem sich Zustände verändert haben. Diese können entweder durch den Trap-verursachenden Befehl oder spätere Befehle verändert worden sein. Dabei muss es u.U. möglich sein, die Trap-auslösende Instruktion bei der Trap Behandlung zu emulieren.
- *Interrupting trap*: Darunter werden externe Interrupt Anforderungen oder Ausnahmen, die nicht in Bezug zu einem Befehl gesetzt werden können, verstanden. Ausnahmen, die deferred Traps ähnlich sind, jedoch nicht die Trap-auslösende Instruktion emulieren können, werden auch unter diese Kategorie gezählt. Zusätzlich zum ET Feld wird auch das PIL Feld des PSR mit dem angeforderten Interrupt Level verglichen, um eine Aussage darüber zu treffen, ob der Trap behandelt wird oder nicht. Der angeforderte Interrupt Level muss größer als der Wert des PIL sein, um behandelt zu werden. Ein Interrupt Level von 15 wird immer behandelt, vorrausgesetzt ET = 1.

Im LEON ist das Standard Trap Modell der SPARC Spezifikation implementiert. Dieses besagt, dass alle Traps precise Traps sein müssen mit folgenden Ausnahmen:

- Ausnahmen, die durch die FPU oder den CP signalisiert werden, dürfen deferred sein.
- Ausnahmen, die es unmöglich machen, den Prozessor wieder in den Zustand zu versetzen, als die Ausnahme verursacht wurde, können interrupting oder deferred sein.
- Von Befehlen unabhängige Ausnahmen sind interrupting Traps.

Traps werden dadurch angezeigt, dass die Hardware einen Eintrag in das *Trap Type*(TT) Feld des *Trap Base Register* (TBR) macht (s. Abb. 2-13). Die 8 Bit des TT Feldes ermöglichen 256 unterschiedliche Traps. Die Hälfte davon ist für Hardware Traps vorgesehen, die andere Hälfte für Software Traps. Software Traps sind Traps, die durch Software ausgelöst werden. Dies geschieht über den Befehl Ticc.

Neben dem TT Feld enthält das TBR auch die *Trap Base Address*(TBA). Die untersten 4 Bit sind Nullen. Das gesamte Register TBR stellt eine Adresse dar. An dieser Adresse befindet sich der *Trap Handler*, also Software Code, der die Behandlung des Traps sicherstellt. Man erkennt leicht, dass alle 256 möglichen Traps einen zusammenhängenden Speicherbereich von 4 kB belegen, dessen Anfang durch die TBA gegeben ist. Dieser Speicherbereich wird als *Trap Table* bezeichnet. Durch die Aufteilung des Speicherbereiches in 4 Befehle pro Trap(die 4 LSB des TBR sind immer Null) ist der Platz, den ein Trap Handler einnehmen könnte sehr gering. Deshalb sind an den jeweiligen Stellen normalerweise neue Einsprungadressen festgelegt (also Sprungbefehle), an denen der eigentliche Trap Handler steht.

Allen Traps sind Prioritäten zugeordnet. Priorität 1 ist die höchste, Priorität 31 die niedrigste Priorität. Sind zum gleichen Zeitpunkt verschiedene Ausnahmen oder Interrupts aktiv, wird der Trap mit der höchsten Priorität bevorzugt behandelt. Dabei wird davon ausgegangen, dass nicht behandelte Interrupt Anforderungen weiter bestehen bleiben und Ausnahmen mit niedrigerer Priorität als der des aktuell Behandelten bei einer erneuten Ausführung des ursächlichen Befehls wieder auftreten (d.h., nicht behandelte Ausnahmen und Interrupt Anforderungen werden der Prioritätenfolge nach abgearbeitet). Die Trap Tabelle des LEON ist in Tabelle 2-6 dargestellt. Die Bedeutung der einzelnen Traps kann in der SPARC V8 Spezifikation bzw. dem LEON Handbuch gefunden werden.

Trap	**TT**	**Pri**	**Description**
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Instruction or data watchpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hadrware_error	0x20	9	register file EDAC error (LEON-FT only)
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception
cp_exception	0x28	11	Co-processor exception
data_access_exception	0x09	13	Access error during load or store instruction
tag_overflow	0x0A	14	Tagged arithmetic overflow
divide_exception	0x2A	15	Divide by zero
interrupt_level_1	0x11	31	Asynchronous interrupt 1
interrupt_level_2	0x12	30	Asynchronous interrupt 2
interrupt_level_3	0x13	29	Asynchronous interrupt 3
interrupt_level_4	0x14	28	Asynchronous interrupt 4
interrupt_level_5	0x15	27	Asynchronous interrupt 5
interrupt_level_6	0x16	26	Asynchronous interrupt 6
interrupt_level_7	0x17	25	Asynchronous interrupt 7
interrupt_level_8	0x18	24	Asynchronous interrupt 8
interrupt_level_9	0x19	23	Asynchronous interrupt 9
interrupt_level_10	0x1A	22	Asynchronous interrupt 10
interrupt_level_11	0x1B	21	Asynchronous interrupt 11
interrupt_level_12	0x1C	20	Asynchronous interrupt 12
interrupt_level_13	0x1D	19	Asynchronous interrupt 13

interrupt_level_14	0x1E	18	Asynchronous interrupt 14
interrupt_level_15	0x1F	17	Asynchronous interrupt 15
trap_instruction	0x80 -	16	Software trap instruction (TA)
	0xFF		

Tabelle 2-6 Trap Table des LEON (Quelle: [L10])

2.2.1.2. Cache System

Der LEON implementiert eine Harvard Architektur. Dies bedeutet, dass getrennte Befehls- und Datenbusse vorhanden sind, die an getrennte Befehls- und Datencaches angeschlossen sind.

Sowohl Befehls- als auch Datencache können auf verschiedene Weise implementiert werden. Dies hängt von der Konfiguration des LEONs bei Synthese ab. Prinzipiell kann zwischen *Direct Mapped Cache* und *Multi-Set Cache* (auch *n-way Set Associativ Cache* genannt) gewählt werden. Der Unterschied zwischen beiden liegt darin, dass bei Multi-Set Caches noch bezüglich der Art und Weise gewählt werden kann, wie die verschiedenen Sets beschrieben werden.

Die Größe jedes Sets (auch die des Direct Mapped Cache) ist zwischen 1 – 64 kB konfigurierbar. Ein Set besteht aus Tag-RAM und Daten-RAM. Im Daten-RAM werden die gecacheten Informationen aufbewahrt. Das Daten-RAM ist in Zeilen von 16 – 32 Bytes unterteilt (also 2 – 8 32-Bit-Worte). Pro Zeile im Daten-RAM existiert auch eine Zeile im Tag-RAM, die dazu benutzt wird, auf Treffer hin zu überprüfen. Abb. 2-15 illustriert eine Tag-RAM Zeile. Diese ist in Befehls- und Datencache identisch.

Bei Implementierung eines Multi-Set Caches können bis zu 4 Sets implementiert werden. Als Auswahlverfahren, in welcher Reihenfolge die Sets beschrieben werden, stehen *Least Recently Used* (LRU) und *Random* zu Verfügung. Bei Implementierung von 2 Sets steht auch noch das Verfahren *Least Recently Replaced* (LRR) zu Verfügung.

Es werden immer nur die Bits und Register implementiert, die in der Konfiguration angegeben sind.

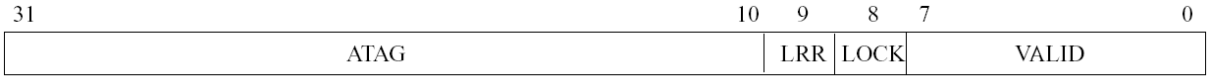


Abb. 2-15 Tag-RAM Zeile (Quelle: [L10])

Bedeutung der Felder:

- *Address tag* (ATAG): Enthält das Tag der Cache Zeile.
- LRR: Wird bei Verwendung des LRR Algorithmus von diesem benutzt.
- LOCK: signalisiert, ob eine Cache Zeile blockiert ist (*locked*).
- VALID: Für jedes 32-Bit-Wort im korrespondierenden Daten-RAM ist ein Bit vorgesehen. Eine 1 signalisiert, dass das zugehörige Wort im Cache enthalten ist.

Befehlscache

Neue Instruktionen werden über den Befehlscache geholt. Enthält dieser nicht den Befehl, der an der angeforderten Adresse steht, wird der Befehl aus dem Speicher geholt, im Cache abgelegt und an die IU weitergereicht. Ist *Instruction Burst Fetch* im *Cache Control Register* (CCR) aktiviert, wird die gesamte Cache Zeile mit den folgenden Befehlen aufgefüllt. Dieser Vorgang wird abgebrochen, wenn ein Sprungbefehl ausgeführt wird. Hintergrund dieser Funktion ist, dass Befehle ihrer chronologischen Abfolge entsprechend linear hintereinander im Speicher abgelegt sind. Ausnahmen davon bilden Sprungbefehle.

Die mit den im Daten-RAM stehenden Befehlen korrespondierenden VALID-Bits werden bei erfolgreichem Lesen aus dem Speicher auf ‚1' gesetzt.

Datencache

Ist ein angefordertes Datum nicht im Datencache vorhanden, wird es aus dem Speicher gelesen. Ein Äquivalent zum *Instruction Burst Fetch* gibt es nicht, da Daten im Allgemeinen nicht in der linearen Reihenfolge wie Befehle abgespeichert sind und benötigt werden.

Im Gegensatz zu Befehlen werden Daten nicht nur geladen, sondern auch abgespeichert. Der im LEON implementierte Daten-Cache benutzt dafür das *Write-Through* Verfahren. Dieses verändert - sollte ein Eintrag für die entsprechende Schreibadresse im Cache vorhanden sein (sog. *Write-Hit*) - sowohl den Cache als auch das Datum im Speicher. Unter der Vorraussetzung, dass keine Fehler beim Schreiben des Speichers auftreten, herrscht Datenkonsistenz zwischen Cache und Speicher. Ist kein Eintrag unter der Schreibadresse im Cache vorhanden (sog. *Write-Miss*), bleibt der Cache unverändert.

Der Daten-Cache enthält noch zusätzlich einen *Write Buffer* (WRB), der aus drei 32-Bit Registern besteht. Diese werden verwendet, um die zu schreibenden Daten kurzzeitig zu speichern, bis sie an der Zieladresse empfangen worden sind. Ein Register speichert die Adresse, die zwei restlichen enthalten den zu schreibenden Wert (zwei Register sind notwendig, wenn Daten vom Typ *double* geschrieben werden).

Spezielle Cachezugriffe

Der LEON implementiert eine Reihe von speziellen Cachezugriffen. Diese sind über die *Load/Store from alternate Space* (LDA/STA) Befehle benutzbar. Diese Befehlstypen sind durch die SPARC V8 Spezifikation vorgegeben. Dabei wird neben der obligatorischen Adresse auch noch ein *Alternate Space Identifier* (ASI) mit dem Befehl angegeben, der z.B. von der Systemsoftware dazu benutzt werden kann, bestimmte geschützte Register zu adressieren. ASI ist ein 8-Bit-Wert, es stehen demnach 256 unterschiedliche ASI zu Verfügung. Die SPARC Spezifikation macht nur Vorschläge, wie diese zu benutzten sind. Der LEON benutzt die unteren 4 Bit, um spezielle Zugriffe auf den Cache zu machen.

ASI	Usage
0x0, 0x1, 0x2, 0x3	Forced cache miss (replace if cacheable)
0x4, 0x7	Forced cache miss (update on hit)
0x5	Flush instruction cache
0x6	Flush data cache
0x8, 0x9, 0xA, 0xB	Normal cached access (replace if cacheable)
0xC	Instruction cache tags
0xD	Instruction cache data
0xE	Data cache tags
0xF	Data cache data

Tabelle 2-7 ASI Gebrauch (Quelle: [L10])

Tabelle 2-7 zeigt die verschiedenen Zugriffe. Bei Verwendung von ASI 0x0 – 0x3 wird cacheintern ein *Cache Miss* erzeugt, unabhängig davon, ob ein passender Eintrag vorhanden ist oder nicht. Ein schon vorhandener Eintrag im Cache wird aktualisiert, ein noch nicht vorhandener wird als neuer Eintrag in den Cache geschrieben.

ASI 0x4 und ASI 0x7 funktionieren ähnlich, mit dem Unterschied, das keine neuen Einträge angelegt werden.
ASI 0x5 und ASI 0x6 löschen den entsprechenden Cache (siehe: Löschen des Cache).
ASI 0x8 – ASI 0xB entspricht der normalen, oben beschriebenen Funktionsweise der Caches.
ASI 0xC – ASI 0xF werden verwendet, um gezielt das Tag-RAM oder das Daten-RAM des jeweiligen Caches zu lesen oder zu beschreiben.

LDA und STA können nur ausgeführt werden, wenn der Prozessor im *Supervisor Mode* arbeitet, d.h. das S-Bit im PSR ist aktiv. Es können nur solche Daten aus externen Bauteilen gecached werden, deren Adressbereich in der Konfiguration des LEON als cachefähig angegeben wurde. Als Standard ist dies der PROM- und der RAM-Bereich.

Löschen des Cache

Befehls- und Daten-Cache können unabhängig voneinander gelöscht werden. Dies geschieht entweder durch einen LDA bzw. STA Befehl mit ASI 0x5 oder ASI 0x6 oder durch Setzen des Bits 21 bzw. Bits 22 des *Cache Control Registers* (CCR). Das Löschen dauert einen Takt pro Cache Zeile. Während dieser Zeit ist der Cache deaktiviert. Danach wird der Cache automatisch wieder in den (Aktivitäts-) Zustand versetzt, der durch das CCR vorgegeben ist.

Cache Control Register (CCR)

Das CCR dient zur Kontrolle des Befehls- und des Datencaches und zum Abfragen der Zustände der Caches. Abb. 2-16 zeigt den Aufbau des CCR.

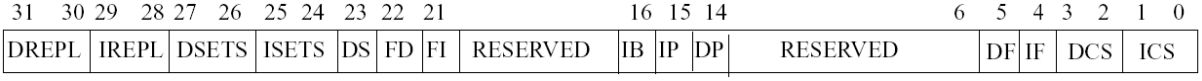


Abb. 2-16 Cache Control Register (CCR) (Quelle: [L10])

Die Felder haben folgende Bedeutungen:

- *Data cache replacement policy* (DREPL): Gibt an, auf welche Weise zwischen den Sets des Daten-Caches gewechselt wird. 00: kein Wechsel (entspricht Direct Mapped Cache), 01: zufällig, 10: least recently replaced (LRR), 11: least recently used (LRU)
- *Instruction replacement policy* (IREPL): Entspricht DREPL für den Befehlscache.
- *Data cache associativity* (DSETS): Gibt die Anzahl der Sets des Datencaches an.
- *Instruction associativity* (ISETS): Entspricht DSETS für den Befehlscache.
- *Data cache snoop enable* (DS): Wenn gesetzt, ist *cache snooping* des Datencaches aktiviert.
- *Flush data cache* (FD): Wenn gesetzt, wird der Datencache gelöscht.
- *Flush instruction cache* (FI): Wenn gesetzt, wird der Befehlscache gelöscht.
- *Instruction burst fetch* (IB): Wenn gesetzt, werden beim Cachen im Befehlscache ganze Cache-Zeilen aus dem Speicher gecached.
- *Instruction cache flush pending* (IP): Ist gesetzt, wenn der Befehlscache gerade gelöscht wird.
- *Data cache flush pending* (DP): Entspricht IP für den Datencache.
- *Data cache freeze on interrupt* (DF): Wenn gesetzt, wird der Datencache automatisch eingefroren, wenn eine Interrupt Anforderung an die IU signalisiert wird. D.h. alle Inhalte des Cache bleiben gespeichert und können erst wieder verändert werden, wenn der Zustand des Caches über Ändern von DCS auf aktiv geändert wird.
- *Instruction cache freeze on interrupt* (IF): Entspricht DF für den Befehlscache. Ein aktiver Zustand wird über Ändern von ICS eingestellt.
- *Data cache state* (DCS): Zeigt den Zustand des Datencaches an und dient zur Änderung desselben. X0: deaktiviert (der Cache arbeitet nicht), 01: eingefroren, 11: aktiviert.
- *Instruction cache state* (ICS): Entspricht DCS für den Befehlscache.

2.2.1.3. AMBA-Bussystem

Im LEON werden zwei verschiedene Busse als on-chip Bussysteme verwendet, die nach der AMBA-Spezifikation arbeiten. Die *Advanced Microcontroller Bus Architecture* (AMBA) Spezifikation definiert drei verschiedene Standards für on-chip Bussysteme. Im Gegensatz zum PCI-Bus sind Daten- und Adressleitungen physikalisch getrennt. Der Unterschied zwischen den beiden verwendeten AMBA-Bussen liegt in Bus-Performance und Komplexität der Buscontroller der einzelnen Peripherie-Komponenten.

AHB-Bus

Der *Advanced High Performance Bus* (AHB) wird für Anwendungen verwendet, die einen hohen Datendurchsatz erfordern. Im LEON wird das AHB-Interface der IU über den AHB-Bus mit dem Memory Controller verbunden. Außerdem ist die APB-Bridge und die *Debug Support Unit* (DSU) angeschlossen (vgl. Abb. 2-8). Das Address-Mapping ist durch die Konstante *ahbslvcfg_tkconfig* im VHDL-Code der Datei „device.vhd“ festgelegt. Als Standard wird das Mapping aus Tabelle 2-8 verwendet.

Address range	Size	Mapping	Module
0x00000000 - 0x1FFFFFFF 0x20000000 - 0x3FFFFFFF 0x40000000 - 0x7FFFFFFF	512 M 512 M 1 G	Prom Memory bus I/O SRAM and/or SDRAM	Memory controller
0x80000000 - 0x8FFFFFFF	256 M	On-chip registers	APB bridge
0x90000000 - 0x9FFFFFFF	256 M	Debug support unit	DSU

Tabelle 2-8 Standard AHB-Mapping (Quelle: [L10])

APB-Bus

Der *Advanced Peripheral Bus* (APB) ist dem AHB-Bus nachgeschaltet. Als Bindeglied dient die APB-Bridge (vgl. Abb. 2-8). Der LEON benutzt den APB-Bus vor allem dazu, Kontroll- und Statusregister der on-chip liegenden Peripherie zu adressieren. Das Mapping des APB-Busses ist durch die Konstante *apbslvcfg_tkconfig* im VHDL-Code der Datei „device.vhd“ festgelegt. Als Standard wird das Mapping aus Tabelle 2-9 verwendet. Die Verwendung der einzelnen Register ist den entsprechenden Komponentenbeschreibungen zu entnehmen.

Address	Register	Address	
0x80000000	Memory configuration register 1	0x800000B0	Secondary interrupt mask register
0x80000004	Memory configuration register 2	0x800000B4	Secondary interrupt pending register
0x80000008	Memory configuration register 3	0x800000B8	Secondary interrupt status register
0x8000000C	AHB Failing address register	0x800000B8	Secondary interrupt clear register
0x80000010	AHB status register		
0x80000014	Cache control register	0x800000C4	DSU UART status register
0x80000018	Power-down register	0x800000C8	DSU UART control register
0x8000001C	Write protection register 1	0x800000CC	DSU UART scaler register
0x80000020	Write protection register 2		
0x80000024	LEON configuration register		
0x80000040	Timer 1 counter register		
0x80000044	Timer 1 reload register		
0x80000048	Timer 1 control register		
0x8000004C	Watchdog register		
0x80000050	Timer 2 counter register		
0x80000054	Timer 2 reload register		
0x80000058	Timer 2 control register		
0x80000060	Scaler counter register		
0x80000064	Scaler reload register		
0x80000070	Uart 1 data register		
0x80000074	Uart 1 status register		
0x80000078	Uart 1 control register		
0x8000007C	Uart 1 scaler register		
0x80000080	Uart 2 data register		
0x80000084	Uart 2 status register		
0x80000088	Uart 2 control register		
0x8000008C	Uart 2 scaler register		
0x80000090	Interrupt mask and priority register		
0x80000094	Interrupt pending register		
0x80000098	Interrupt force register		
0x8000009C	Interrupt clear register		
0x800000A0	I/O port input/output register		
0x800000A4	I/O port direction register		
0x800000A8	I/O port interrupt register		

Tabelle 2-9 On-Chip Register / APB-Mapping (Quelle: [L10])

2.2.1.4. Interrupt Controller

Moderne Prozessoren arbeiten mit Prioritäten in der Interruptverarbeitung. D.h. dass durch Systemsoftware den verschiedenen Interrupts, die in einem System auftreten können, auch verschiedene Stufen der „Wichtigkeit“ zugeordnet werden. Dies hat den Vorteil, dass bei Anforderungen mehrerer Interrupts zur gleichen Zeit zuerst der abgearbeitet wird, der die

höchste Priorität hat. Der LEON arbeitet mit zwei Prioritäts-Leveln: Level 1 und Level 0. Level 1 wird bevorzugt vor Level 0 abgearbeitet. Er hat die höhere Priorität. Innerhalb eines Levels wird dem Interrupt der Vorzug gegeben, der die höchste Nummerierung aufweist. D.h. wenn zwei Interrupt Anforderung mit Level 1 an den Interrupt Controller signalisiert werden – z.B. Nr. 15 und Nr. 6 -, wird der Interrupt Nr. 15 zuerst an die IU weitergegeben.

Immer der Interrupt mit der höchsten Priorität wird an die IU weitergeleitet.

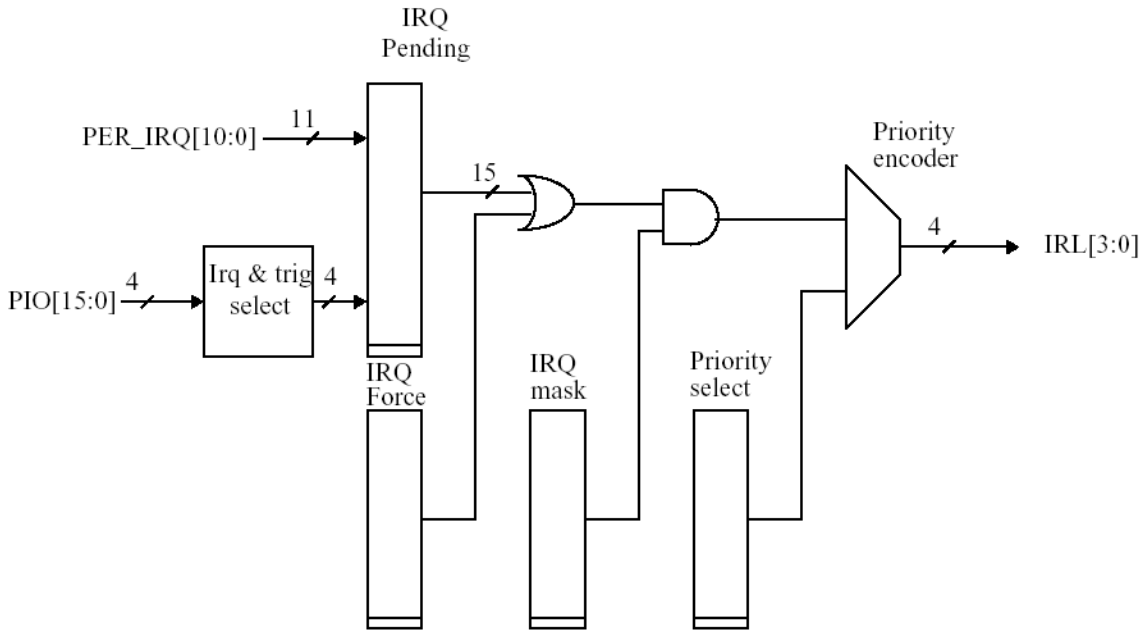


Abb. 2-17 Blockdiagramm des Interrupt Controller (Quelle: [L10])

Abb. 2-17 zeigt den Aufbau des Interrupt Controllers. Insgesamt 15 verschiedene Interrupt Leitungen werden durch den Interrupt Controller bewertet. Davon können vier über die parallele I/O-Schnittstelle des LEON angeschlossen werden (s. Kapitel 2.2.1.7). Von den restlichen 11 sind weitere vier Interrupts in der Standard-Konfiguration noch nicht vergeben und können von anwenderspezifischen Komponenten belegt werden.

Durch vier Kontroll-Register lässt sich der Controller einstellen. *IRQ-Pending* speichert eingehende Interrupt Anforderungen. Mit *IRQ-Force* lässt sich ein Interrupt manuell erzwingen. Enthält eines dieser beiden Register eine 1, wird der entsprechende Eintrag mit dem *IRQ-Mask* und *Priority Select* Register verglichen. Dort sind Informationen über den Level des Interrupts enthalten und ob dieser Interrupt aktiviert ist. Das vierte Register ist in Abb. 2-17 nicht abgebildet. Es heißt *IRQ-Clear* Register und wird dazu benutzt, das *IRQ-Pending* Register zu löschen. Eine Belegung der Register ist in dem Benutzerhandbuch des LEONs nachzulesen.
Der *Priority Encoder* selektiert in Abhängigkeit von IRQ-Level und IRQ-Nummer den Interrupt, der an die IU weitergeleitet wird. Der Block *IRQ&Trig-Select* ist unter dem Modul für die Parallel Schnittstelle (PIO) definiert. Der Block enthält Informationen darüber, welche PIO-Signale als Interrupts verwendet werden sollen und auf welche Flanke getriggert werden soll.
Tabelle 2-10 zeigt die Belegung der 15 möglichen Interrupts. Interrupt Nr. 10 ist für einen zweiten, optionalen Interrupt Controller reserviert, durch den zusätzlich 32 Interrupt-Leitungen zu Verfügung stehen. Eine genauere Beschreibung dieses Zusatz Controllers ist in der Dokumentation des LEON zu finden.

Interrupt	Source
15	user defined
14	user defined
13	user defined
12	user defined
11	DSU trace buffer
10	Second interrupt controller
9	Timer 2
8	Timer 1
7	Parallel I/O[3]
6	Parallel I/O[2]
5	Parallel I/O[1]
4	Parallel I/O[0]
3	UART 1
2	UART 2
1	AHB error

Tabelle 2-10 Interrupt Tabelle (Quelle: [L10])

2.2.1.5. Timer

Prozessoren verwenden Timer, um eine definierte bzw. einstellbare Zeitspanne zu messen, nach der u.U. eine Routine gestartet wird. Timer arbeiten unabhängig von der IU. Um Routinen unabhängig von gerade laufenden Programmen ausführen zu können, signalisieren die Timer das Ende einer Zeitspanne durch einen Interrupt.
Der LEON implementiert zwei 24-Bit Timer und einen 24-Bit *Watchdog*. Der Unterschied zwischen Timer und Watchdog besteht darin, dass die Timer einen Zählerstand von Null durch eine Interrupt Anforderung ausdrücken, eigene Register für Startwerte haben und deaktiviert werden können. Der Watchdog hat ein externes Ausgangssignal (off-chip), ist immer aktiviert und kann keinen automatischen Wiederladewert zugewiesen bekommen.

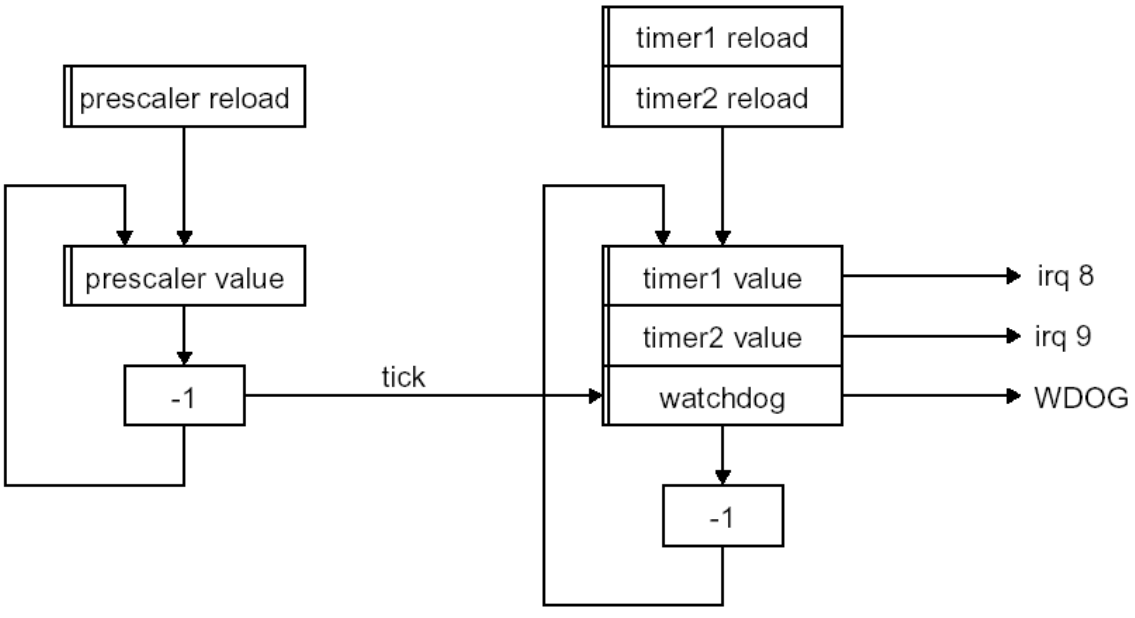


Abb. 2-18 Blockdiagramm der Timer (Quelle: [L10])

Das Blockdiagramm der Timer (siehe Abb. 2-18) verdeutlicht die Funktionsweise der beiden Timer und des Watchdogs. Für alle drei wird ein gemeinsamer Zählertakt (tick) verwendet. Dieser ist über das *Prescaler Reload* Register einstellbar. Der Wert dieses Registers wird in das *Prescaler Value* Register geladen und dort mit jedem Systemtakt dekrementiert. Bei einem Wert von Null wird ein Zähltaktsignal (tick) erzeugt und das Prescaler Value Register wieder mit dem Prescaler Reload Register geladen. Es können Werte zwischen 3 und 1023 eingestellt werden. Welchem Zähltakt für die Timer das entspricht, ist abhängig vom Systemtakt.
Die Werte der Timer und des Watchdog sind in den Registern *Timer1 Value* bzw. *Timer2 Value* und *Watchdog Value* abgelegt. Wie die Timer Register sich bei einem Zähltaktsignal verhalten, ist abhängig von den in Abb. 2-18 nicht dargestellten *Timer1 Control* bzw. *Timer2 Control* Registern. Die drei LSB dieser Register legen das Verhalten des entsprechenden Timers fest.

- Bit 0: *Enable* (EN). Aktiviert den Timer, wenn es auf 1 gesetzt ist.
- Bit 1: *Reload Counter* (RL). Wenn dieses Bit gesetzt ist und der Timer auf Null dekrementiert worden ist, wird er automatisch mit dem Wert des entsprechenden *Reload* Registers nachgeladen.
- Bit 2: *Load Counter* (LC). Soll das Timer Value Register manuell geladen werden, muss dieses Bit mit einer Eins beschrieben werden.

Ist EN=1, wird ein Timer bei einem Zählertaktsignal dekrementiert. Dasselbe passiert mit dem Watchdog Value Register, wenn ein Wert geladen ist. Ist der Inhalt eines dieser Register Null, wird das entsprechende Ausgangssignal aktiviert: Timer1 liegt an IRQ 8, Timer2 an IRQ9 und der Watchdog an dem off-chip Signal WDOG. Ein Timer wird in Abhängigkeit von RL automatisch mit dem Inhalt des *Timer Reload* Register geladen oder manuell mit LC=1. Der Wert des Watchdog Value Registers muss über einen Schreibzugriff auf dieses Register geladen werden.

2.2.1.6. UARTs

Um mit anderen Geräten kommunizieren zu können, die keinen hohen Datendurchsatz benötigen und eventuell über längere Kabel verbunden sind, stellt der LEON zwei identische UARTs zu Verfügung. Diese generieren aus 8-Bit Worten einen seriellen Datenstrom mit oder ohne Parität. Die Umsetzung auf einen elektrischen Standard (z.B. RS232) geschieht außerhalb des LEON durch spezielle Treiberbausteine.

Abb. 2-19 zeigt das Blockschaltbild einer UART. Die Baudrate ist einstellbar durch einen 12-Bit Taktteiler. Dieser lässt sich durch das *UART Scaler Reload Register* einstellen. Optional kann statt des Systemtaktes auch ein externer Takt geteilt werden, der an die parallele Schnittstelle (PIO[3]) angeschlossen ist.
Zu Testzwecken kann eine UART in den *Loop Back* Modus geschaltet werden. Dann wird intern TXD auf RXD und RTSN auf CTSN geschaltet.
Der LEON unterstützt Handshaking zwischen Sender und Empfänger durch die Signale RTSN und CTSN.

Die Software kann die UARTs so einstellen, dass sie das Senden oder Empfangen eines Zeichens durch Generierung von Interrupts anzeigen. Dabei wird pro UART ein Interrupt verwendet (vgl. Kapitel 2.2.1.4).

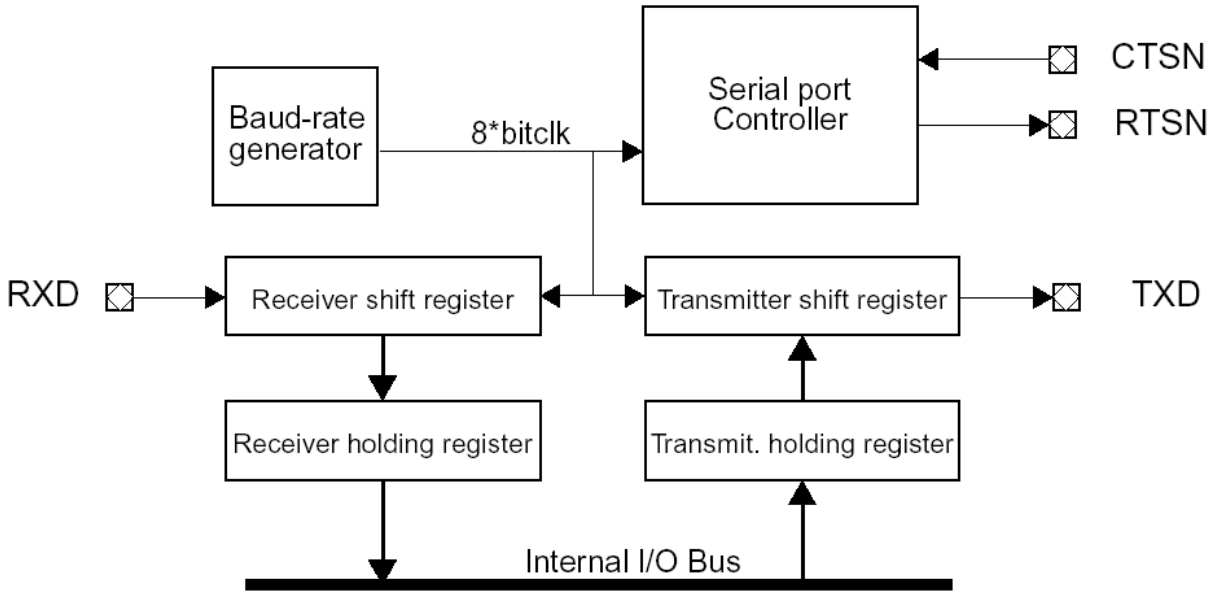


Abb. 2-19 Blockdiagramm UART (Quelle: [L10])

Sender

Daten, die übertragen werden sollen, müssen in das *Transmitter Holding Register* geschrieben werden. Dies geschieht über Beschreiben der 8 LSB des *UART Data Registers* (0x80000070 bzw. 0x80000080). Ist die UART bereit zum Senden, werden die unteren acht LSB dieses Registers in das *Transmitter Shift Register* geladen und seriell über TXD gesendet. Dabei wird zuerst ein Startbit gesendet, gefolgt von den acht Datenbits (LSB zuerst). Anschließend wird optional ein Paritätsbit gesendet und zum Schluss ein Stopbit. Abb. 2-20 zeigt beide Möglichkeiten.
Ist das Hardware Handshaking aktiviert, muss CTSN aktiv sein (d.h. Low), damit ein Sendevorgang gestartet wird. CTSN wird mit dem RTSN Ausgang des (externen) Empfängers verbunden.
Ist die Interrupt Generierung für Senden aktiviert, wird ein Interrupt signalisiert, wenn das Transmitter Holding Register in das Transmitter Shift Register kopiert wird. Durch den Interrupt wird „mitgeteilt“, dass das nächste zu sendende Zeichen in das Transmitter Holding Register geschrieben werden kann.

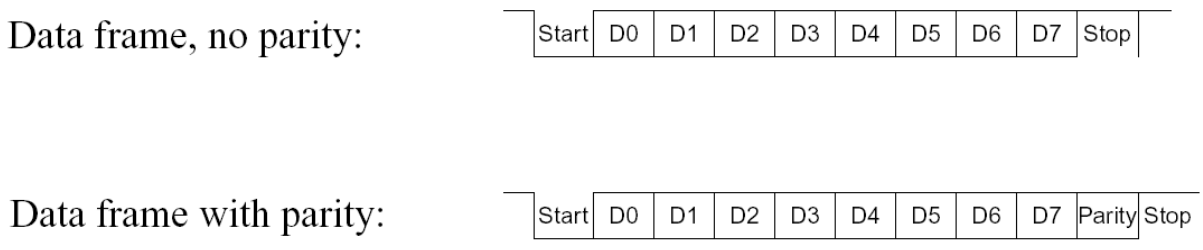


Abb. 2-20 Sendeformat der UART (Quelle: [L10])

Empfänger

Ist der Empfänger aktiviert und registriert an RXD ein Startbit (d.h. ein Übergang von 1 auf 0), wird das empfangene Zeichen (8-Bit) in dem *Receiver Shift Register* gespeichert und

anschließend in das *Receiver Holding Register* geschrieben. Dies kann gelesen werden durch einen Lesezugriff auf das UART Data Register. Die acht LSB enthalten das gelesene Zeichen. Beim Empfang wird auch überprüft, ob das Format der gesendeten Information korrekt ist und ob die Parität stimmt. Tritt ein Fehler auf, wird dies im *UART Status Register* gespeichert. Werden empfangene Zeichen nicht schnell genug ausgelesen, kann es passieren, dass Informationen verloren gehen. Enthält sowohl das Receiver Holding Register als auch das Receiver Shift Register noch nicht gelesene Zeichen, wird bei erneutem Empfang eines Zeichens das letztere Register überschrieben und das *overrun* Bit (OV) im UART Status Register gesetzt.
Wenn Handshaking aktiviert ist, zieht die UART RTSN ab Erkennen eines Startbits solange auf High, bis das empfangene Zeichen aus dem Receiver Holding Register gelesen worden ist. Dann wird RTSN wieder auf Low gezogen, um dem (externen) Sender mitzuteilen, dass das nächste Zeichen gesendet werden kann.
Ist die Interrupt Generierung für Empfangen aktiviert, wird ein Interrupt erzeugt, wenn das Receiver Shift Register in das Receiver Holding Register geschrieben wird. Dies signalisiert, dass ein Zeichen gelesen wurde.

UART Register

Jede UART hat ihren eigenen Registersatz. Die Adressierung ist in Kapitel 2.2.1.3 beschrieben.
Das UART Scaler Reload Register legt die Baudrate mit den 12 LSB fest. Alle anderen Bits sind unbenutzt.
Das UART Data Register benutzt nur die unteren 8 Bit. Dort werden entweder Zeichencodes hingeschrieben, um sie zu senden, oder empfangene Zeichen gelesen.
Mit dem *UART Control Registern* lassen sich die UARTs einstellen. Abb. 2-21 zeigt den Aufbau dieses Registers.

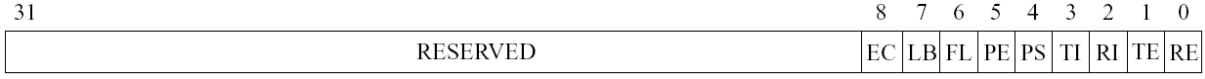


Abb. 2-21 UART Control Register (Quelle: [L10])

- *External Clock* (EC): Wenn gesetzt, wird der externe Takt an PIO(3) zur Erzeugung des Bittaktes benutzt.
- *Loop Back* (LB): Wenn gesetzt, wird der *Loop Back* Modus aktiviert.
- *Flow Control* (FL):Wenn gesetzt, ist das Hardware Handshaking aktiviert.
- *Parity Enable* (PE): Wenn gesetzt, wird beim Senden ein Paritätsbit erzeugt und beim Empfangen die Parität überprüft.
- *Parity Select* (PS): Schaltet zwischen *even* und *odd parity* um (1=*odd*, 0=*even*).
- *Transmitter Interrupt Enable* (TI): Wenn gesetzt, ist Interrupt Generierung beim Senden aktiviert.
- *Receiver Interrupt Enable* (RI): Wenn gesetzt, ist Interrupt Generierung beim Empfangen aktiviert.
- *Transmitter Enable* (TE): Wenn gesetzt, ist der Sender aktiviert.
- *Receiver Enable* (RE): Wenn gesetzt, ist der Empfänger aktiviert.

Das UART Status Register enthält Informationen über Zustand der UART und Fehler beim Übertragen. Die Belegung des Registers ist in Abb. 2-22 dargestellt.

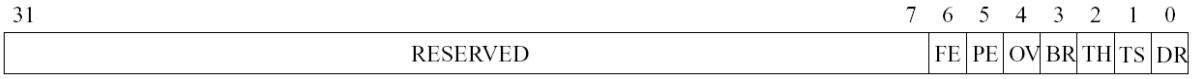


Abb. 2-22 UART Status Register (Quelle: [L10])

- *Framing Error* (FE): Das empfangene Datenformat entsprach nicht dem Soll.
- *Parity Error* (PE): Zeigt einen Paritätsfehler beim Empfang an.
- *Overrun* (OV): Zeigt an, dass Informationen wegen Überschreiben verloren gegangen sind.
- *Break Received* (BR): Zeigt eine Unterbrechung beim Empfangen an.
- *Transmitter Hold Register Empty* (TH): *Trans. Holding* Register ist leer.
- *Transmitter Shift Register Empty* (TS): *Trans. Shift* Register leer.
- *Data Ready* (DR): Zeigt an, dass ein empfangenes Zeichen im *Receiver Holding* Register steht.

2.2.1.7. Parallele Schnittstelle

Der LEON stellt eine 32-Bit breite, parallele Schnittstelle zu Verfügung (*Parallel I/O Port* (PIO)). Allerdings sind diese 32 Bit nur dann ausnutzbar, wenn alle angeschlossenen Speichertypen mit einer Wortbreite von maximal 16 Bit arbeiten und kein SDRAM verwendet wird. Dies ist für die verwendeten Boards nicht der Fall. Das AVNET-Board ist mit SDRAM in 64-Bit Konfiguration bestückt (s. Kapitel 2.1.1.1) und das entwickelte PMC-Board mit SRAM in einer 32-Bit Kombination (s. Kapitel 3.1.4.6). Deshalb bleiben nur die unteren 16 Bit nutzbar (vgl. [L10] Kapitel 6.6).
Von diesen 16 Bit ist jedes Bit während der Laufzeit individuell konfigurierbar, ob es als Ausgang oder Eingang verwendet wird. Dies geschieht über die unteren 16 Bit des *I/O Port Direction Register*. Ein gesetztes Bit entspricht der Benutzung als Ausgang. Um off-chip Pins zu reduzieren, verwendet das LEON-Design bei Benutzung der beiden UARTs oder eines externen Boot PROM den PIO als Schnittstelle dafür. Tabelle 2-11 zeigt die Belegung der einzelnen Ports. Dies schränkt die Möglichkeiten der Verwendung des PIO für anwenderspezifische Zwecke sehr ein.
Vier der Signale können als externe Interrupt Signale benutzt werden (vgl. Kap. 2.2.1.4). Ob und welche Ports als Interrupts benutzt werden sollen, und wie das Eingangssignal getriggert wird, um einen Interrupt auszulösen, wird im *I/O Port Interrupt Configuration Register* festgelegt. Dieses Register ist in Abb. 2-23 dargestellt. Im LEON sind die Interrupts 4 bis 7 für den PIO reserviert. Dabei ist der Port, der durch das Feld ISEL0 im I/O Port Interrupt Configuration Register bestimmt wird, dem IRQ 4 zugeordnet, Feld ISEL1 IRQ 5 usw.

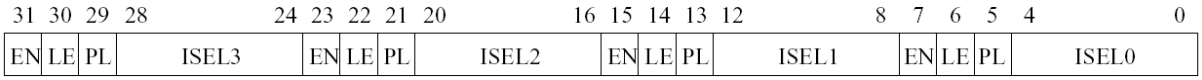


Abb. 2-23 I/O Port Interrupt Configuration Register (Quelle: [L10])

- *I/O Port Select* (ISELn): Legt fest, welcher I/O Port benutzt wird.
- *Polarity* (PL): Wenn gesetzt, wird ein Interrupt bei High Pegel am Eingang ausgelöst bzw. bei einer positiven Flanke (siehe LE).
- *Level/Edge Triggered* (LE): Wenn gesetzt, wird das Eingangssignal auf Flanke getriggert, sonst auf Pegel.
- *Enable* (EN): Wenn gesetzt, ist die Interrupt Generierung aktiviert.

I/O port	Function	Type	Description	Enabling condition
PIO[15]	TXD1	Output	UART1 transmitter data	UART1 transmitter enabled
PIO[14]	RXD1	Input	UART1 receiver data	-
PIO[13]	RTS1	Output	UART1 request-to-send	UART1 flow-control enabled
PIO[12]	CTS1	Input	UART1 clear-to-send	-
PIO[11]	TXD2	Output	UART2 transmitter data	UART2 transmitter enabled
PIO[10]	RXD2	Input	UART2 receiver data	-
PIO[9]	RTS2	Output	UART2 request-to-send	UART2 flow-control enabled
PIO[8]	CTS2	Input	UART2 clear-to-send	-
PIO[4]	Boot select	Input	Internal or external boot prom	-
PIO[3]	UART clock	Input	Use as alternative UART clock	-
PIO[1:0]	Prom width	Input	Defines prom width at boot time	-

Tabelle 2-11 Vordefinierte Belegung des PIO (Quelle: [L10])

2.2.1.8. Memory Controller

An den LEON können off-chip verschiedene Speichertypen angeschlossen werden. Der LEON unterstützt die Verwendung von PROM, SRAM, SDRAM und Memory gemappte I/O-Bausteine.
Um die verschiedenen Komponenten anzusteuern, ist ein Memory Controller implementiert. Dieser dekodiert die Signale auf dem AHB-Bus und übersetzt sie in ein adäquates Format. Dabei sind alle externen Komponenten an den gleichen Daten- und Adressbus angeschlossen. Chip Select (CS) und andere Steuersignale für die jeweiligen Komponenten sind getrennt vorhanden.
Abb. 2-24 zeigt schematisch das Memory Interface des LEON. Durch die Anzahl der vorhandenen CS-Signale lässt sich die Anzahl der unterstützten Speicherbänke bestimmen:

- 2 PROM-Bänke
- 1 I/O Bank
- 5 SRAM Bänke
- 2 SDRAM-Bänke

Der AHB-Bus hat einen möglichen Adressbereich von 4 GB. Von diesen 4 GB dekodiert der Memory Controller die unteren 2 GB. Tabelle 2-12 zeigt die Standard-Zuordnung des LEON. Sind SRAM und SDRAM angeschlossen, wird SRAM ab 0x40000000 und SDRAM ab 0x60000000 adressiert.

Address range	Size	Mapping
0x00000000 - 0x1FFFFFFF	512 M	Prom
0x20000000 - 0x3FFFFFFF	512M	I/O
0x40000000 -0x7FFFFFFF	1 G	SRAM/SDRAM

Tabelle 2-12 Adressen-Zuordnung des Memory Controllers (Quelle: [L10])

Der Memory Controller dekodiert anhand der 4 MSB AHB-Adressbits die adressierte Komponente und aktiviert das entsprechende CS-Signal. Die Adressierung auf dem AHB-Bus unterscheidet nicht zwischen verschiedenen Bänken. Diese Differenzierung geschieht im Memory Controller. Der PROM-Adressbereich wird dafür unabhängig von der PROM-Größe halbiert. Der Adressbereich 0x00000000 bis 0x0FFFFFFF wird Bank 0 zugeordnet, der Adressbereich 0x10000000 bis 0x1FFFFFFF Bank 1. Ist die Option eines internen Boot PROMs aktiviert, kann nur ein externes PROM als Bank 1 ab Adresse 0x10000000 adressiert werden.
Die Adressbereiche der verschiedenen RAM-Bänke ist durch die Größe der Einzelbausteine gegeben. Diese muss durch Systemsoftware überprüft und in die Konfigurationsregister des Memory Controllers geschrieben werden.

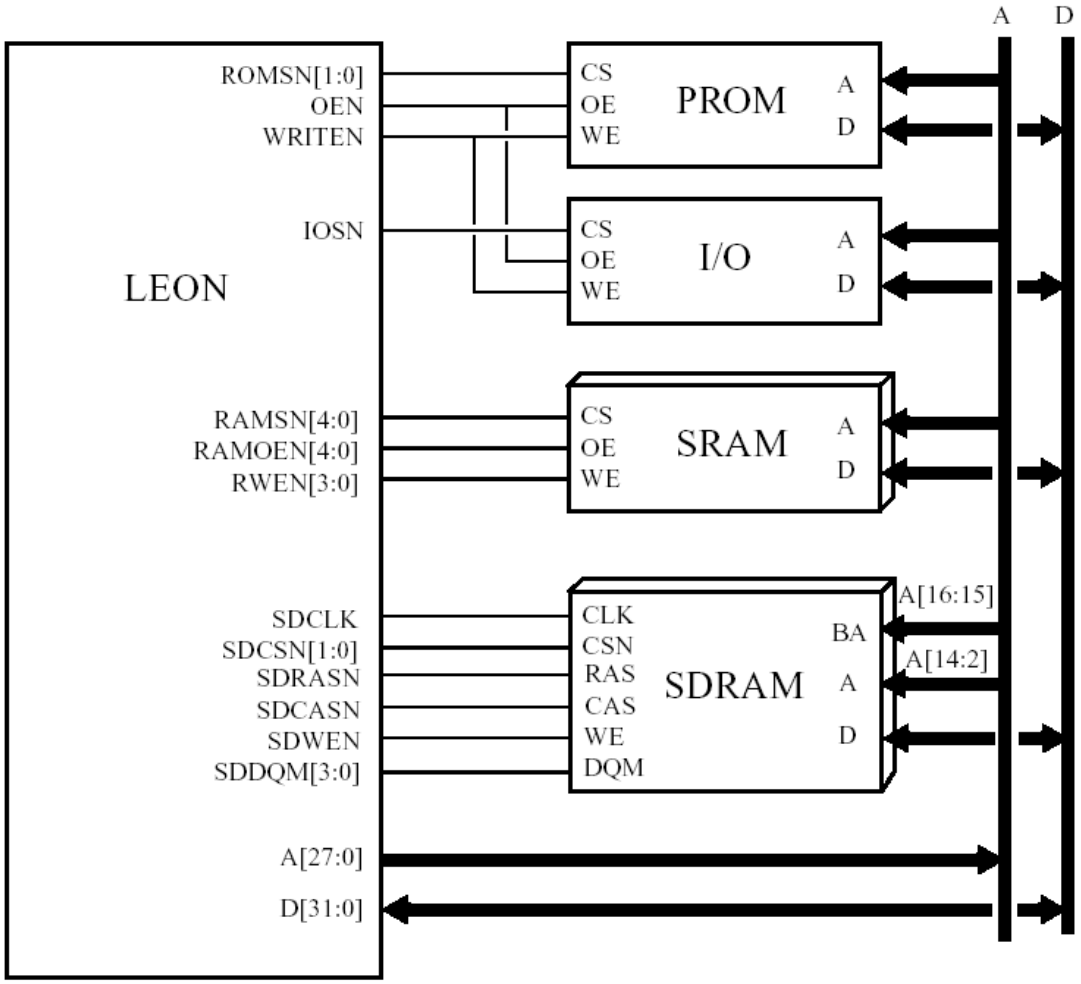


Abb. 2-24 Memory Interface des LEON (Quelle: [L10])

Der Memory Controller wird durch die *Memory Configuration Register* 1 – 3 (MCFG1 -3) konfiguriert. MCFG1 konfiguriert PROM- und I/O-Zugriffe, MCFG2 SRAM- und SDRAM-Zugriffe. MCFG3 wird verwendet, um den Abstand zwischen Refresh Zyklen bei Verwendung von SDRAM zu steuern. Die genaue Beschreibung der Register ist der Beschreibung des LEON zu entnehmen.

2.2.1.9. Debug Support

Um Programme während der Abarbeitung im laufenden Betrieb zu debuggen, stellt der LEON eine *Debug Support Unit* (DSU) zu Verfügung. Über diese kann der Nutzer alle Prozessorregister und Cacheinhalte lesen, wenn der Prozessor in Debug Mode geschaltet ist. Sie kann auch dazu verwendet werden, andere Komponenten über den AHB-Bus zu adressieren. Außerdem ist ein Puffer, der sog. Trace Buffer, vorhanden, der wahlweise die letzten abgearbeiteten Befehle oder Übertragungen auf dem AHB-Bus speichert.

Um die DSU zu verwenden, ist eine einfache UART im LEON implementiert: der *DSU Communication Link* (DCL). Dieser kann z.B. über spezielle Debug Software angesteuert werden. Ob eine DSU implementiert wird, kann durch die Konfigurationseinstellungen des LEONs während der Synthese bestimmt werden. In Abb. 2-25 sind DSU, Trace Buffer und DCL schematisch dargestellt.

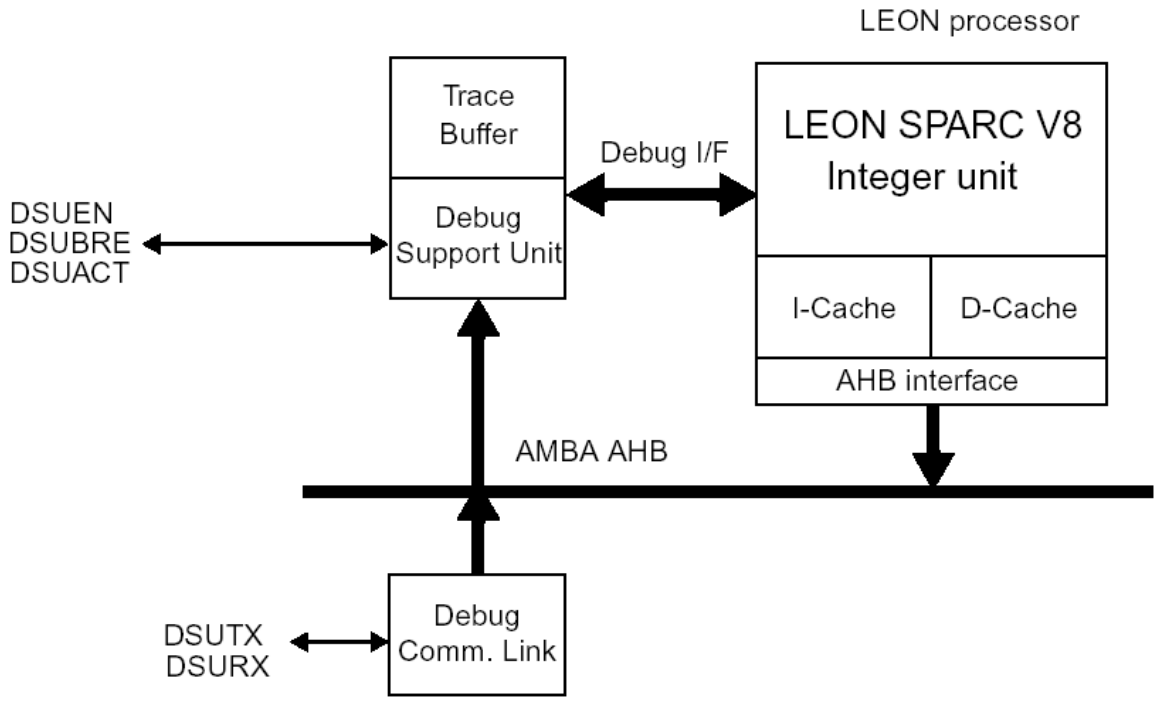


Abb. 2-25 DSU und DCL (Quelle: [L10])

Die externen Signale (off-chip) und ihre Bedeutung sind:

- DSU enable (DSUEN): Aktiviert die DSU.
- DSU break (DSUBRE): Setzt den Prozessor in Debug Mode.
- DSU act (DSUACT): Zeigt an, dass der Prozessor im Debug Mode ist.
- DSUTX: Ist die Sendeleitung des DCL.
- DSURX: Ist die Empfangsleitung des DCL.

Eine Beschreibung der zugrunde liegenden Kontroll- und Statusregister von DSU, Trace Buffer und DCL ist in der Bedienungsanleitung des LEONs zu finden.

Debug Support Unit (DSU)

Die DSU kann über einen 2 MB großen Adressraum auf dem AHB-Bus adressiert werden. Tabelle 2-13 zeigt die Adress-Zuordnung der DSU und des DCL. Die DSU-Kontroll-Register können immer angesprochen werden. Auf den Trace Buffer kann nur zugegriffen werden, wenn dieser aktiviert ist. Der Debug Mode kann aktiviert werden durch:

- Traps
- Traps, die den Prozessor in Error Mode setzen
- IU Watchpoint Treffer
- Steigende Flanke an DSUBRE
- Hardware Breakpoint durch DSU
- Spezielle Anweisung durch DSU
- Single Step Anweisungen der DSU

In allen Fällen muss DSUEN aktiv sein. Dann wird die Abarbeitung des Programms gestoppt, PC und nPC zwischengespeichert, IU Timer und Watchdog werden optional gestoppt und DSUACT wird aktiviert. Das Programm wird weiter ausgeführt, wenn DSUEN deaktiviert wird.

Die DSU enthält zwei Breakpoint Register. Es kann die Adresse von Befehlen oder die Speicheradresse als Triggerbedingung verwendet werden, um den Debug Mode auszulösen und/oder den Trace Buffer einzufrieren.

Trace Buffer

Der Trace Buffer speichert entweder die letzten ausgeführten Befehle, die letzten AHB-Übertragungen oder eine Mischung aus beidem. Dazu steht ihm ein durch das VHDL Modell festgelegter Speicherplatz zu Verfügung (Standard 128 kB), der in Zeilen von 128 Bit (= 16 Byte) aufgeteilt ist. Jede Zeile enthält die zu speichernde Information zu einem Zeitpunkt. Die genaue Belegung dieser Zeilen ist im Manual des LEONs dokumentiert.

DSU Communication Link (DCL)

Im Gegensatz zu den zwei regulären LEON UARTs ist der DCL direkt an den AHB-Bus angeschlossen und kann dort als Master agieren. D.h. über den DCL kann jede beliebige Adresse auf dem AHB-Bus gelesen oder beschrieben werden. Dadurch ist auch die Adressierung der DSU (s. Tabelle 2-13) gewährleistet. Da die UART byteweise arbeitet, ist ein einfaches Protokoll implementiert, um 32-Bit Adresse und Datum zu übertragen. Die Kontroll- und Statusregister des DCL entsprechen einer eingeschränkten Version der Register der regulären UARTs, sind aber genau wie diese über den APB-Bus adressierbar.

Verwendung

Im Allgemeinen ist die DSU dazu da, Programme im Betrieb zu debuggen. Dabei ist die gebräuchlichste Vorgehensweise entweder eine Abarbeitung des Programms in Einzelschritten, ein Festlegen von Adressen, an denen gestoppt wird, oder eine Kombination aus beidem. Auch das Stoppen von Programmen und Debuggen bei Auftreten von Traps ist sinnvoll, da Traps im Allgemeinen in Ausnahmesituationen oder durch Interrupts ausgelöst werden.

Um das Debugging komfortabler zu machen, ist normalerweise eine Software auf einem externen Rechner installiert, mit der der Anwender auf den DCL zugreift. Die Software wird in aller Regel durch abstrakte Befehlssätze gesteuert und zerlegt diese in hardwarespezifische Anweisungen für die DSU. Gaisler Research stellt dazu einen sog. *DSU Monitor* zu Verfügung.

Sind sowohl DSUEN als auch DSUBRE beim Systemstart schon aktiviert, kann die DSU benutzt werden, um darüber den Bootvorgang zu bewerkstelligen. Dies bedeutet, dass die Initialisierung des Prozessors und seiner Komponenten von extern gemacht wird, um dann Anwenderprogramme zu laden und zu debuggen. Dies ist für Testzwecke sinnvoll, da dadurch ein Boot PROM nicht benötigt wird oder hierdurch neu konfiguriert werden kann.

Address	Register
0x800000c4	DSU UART status register
0x800000c8	DSU UART control register
0x800000cc	DSU UART scaler register
0x90000000	DSU control register
0x90000004	Trace buffer control register
0x90000008	Time tag counter
0x90000010	AHB break address 1
0x90000014	AHB mask 1
0x90000018	AHB break address 2
0x9000001C	AHB mask 2
0x90010000 - 0x90020000	Trace buffer
..0	Trace bits 127 - 96
...4	Trace bits 95 - 64
...8	Trace bits 63 - 32
...C	Trace bits 31 - 0
0x90020000 - 0x90040000	IU/FPU register file
0x90080000 - 0x90100000	IU special purpose registers
0x90080000	Y register
0x90080004	PSR register
0x90080008	WIM register
0x9008000C	TBR register
0x90080010	PC register
0x90080014	NPC register
0x90080018	FSR register
0x9008001C	DSU trap register
0x90080040 - 0x9008007C	ASR16 - ASR31 (when implemented)
0x90100000 - 0x90140000	Instruction cache tags
0x90140000 - 0x90180000	Instruction cache data
0x90180000 - 0x901C0000	Data cache tags
0x901C0000 - 0x90200000	Data cache data

Tabelle 2-13 AHB-Address-Mapping von DSU und DCL (Quelle: [L10])

2.2.1.10. VHDL Struktur

Der LEON wird durch einen VHDL-Sourcecode mit einem Umfang von ca. 58000 Zeilen beschrieben, der von der Firma Gaisler Research zu Verfügung gestellt wird. Von diesen 58000 Zeilen entfallen ca. 3500 auf eine Testbench, ca. 9200 auf Dateien, die technologieabhängige Zellen instantiieren, und 25000 auf die Beschreibung verschieden großer Hardware-Multiplizierer. Die VHDL-Codes lassen sich in vier Kategorien unterteilen:

Konfigurations-Dateien

Konfigurations-Dateien sind als Packages deklariert und legen Konstanten, Einstellungen des LEON und verschiedene Datentypen fest. Auch Komponenten werden zum Teil in solchen Packages deklariert. Tabelle 2-14 gibt einen Überblick über diese Packages. Von diesen müssen nur TARGET und DEVICE vom Anwender zur Konfiguration des LEON verändert werden.

Package Name	Datei Name	Funktion
TARGET	target.vhd	Typendeklarationen, vordefinierte Konfigurationen
DEVICE	device.vhd	Festlegung der benutzten Konfiguration
CONFIG	config.vhd	Berechnung verschiedener Konstanten
SPARCV8	sparcv8.vhd	Befehlssatz (nach SPARC V8)
IFACE	iface.vhd	Typen Deklarationen
MACRO	macro.vhd	Funktionen
AMBA	amba.vhd	Typendeklaration für den AMBA-Bus
AMBACOMP	ambacomp.vhd	Deklaration der Komponenten für den AMBA-Bus
FPULIB	fpu.vhd	Deklaration der FPU-Typen

Tabelle 2-14 Packages zur Konfiguration

Technologie Zuordnung

Über eine Reihe von Packages werden technologieabhängige Elemente erzeugt. Wichtig ist das für die Synthese. Dies betrifft die Instantiierung von RAM und Registern und das Erzeugen der I/O-Pads. Dieses Mappen auf die Zieltechnologie geht immer zuerst über das Package TECH_MAP. Hier wird anhand der Konfigurationseinstellungen zwischen den zu Verfügung stehenden Technologien gewählt. Es kann auch technologieunabhängig gearbeitet werden. Dies ist zur funktionalen Simulation sinnvoll. Andere Packages enthalten dann die spezifischen Instantiierungen. Folgende Technologien werden unterstützt:

- Atmel ATC18, ATC25 und ATC35
- UMC/FS90AB
- TSMC 0.25 um
- UMC 0.18 um
- Xilinx Virtex
- Xilinx Virtex2
- Actel Axcellerator
- Actel Proasic

LEON-Funktionen

Funktionale VHDL-Dateien sind Source-Codes, die die Funktionalität des LEON beschreiben. Dabei werden die Informationen der Konfigurationsdateien benutzt. Das Top-Level heißt entweder „leon“ oder „leon_pci“ in Abhängigkeit davon, ob eine AMBA-PCI-Bridge verwendet wird. Dabei stellt Gaisler Research nur das Port-Interface für diese zu Verfügung, die Bridge selber ist **nicht** als Modul vorhanden. Eine Übersicht über die weitere Hierarchie des VHDL-Cores liefern Abb. 2-26 und Abb. 2-27, die vom Autor selbst erstellt wurden, um einen besseren Überblick zu ermöglichen.

Testbench

Gaisler Research stellt eine Testbench zu Verfügung, mit der der LEON getestet werden kann. Da der LEON nur mit Speicher arbeiten kann, wird wahlweise ein SRAM, ROM oder SDRAM emuliert. Die Testbench erlaubt es, während des Simulationsprozesses die abgearbeiteten Befehle zu Deassemblieren. Auf diese Weise kann die Abarbeitung eines Programms verfolgt werden.

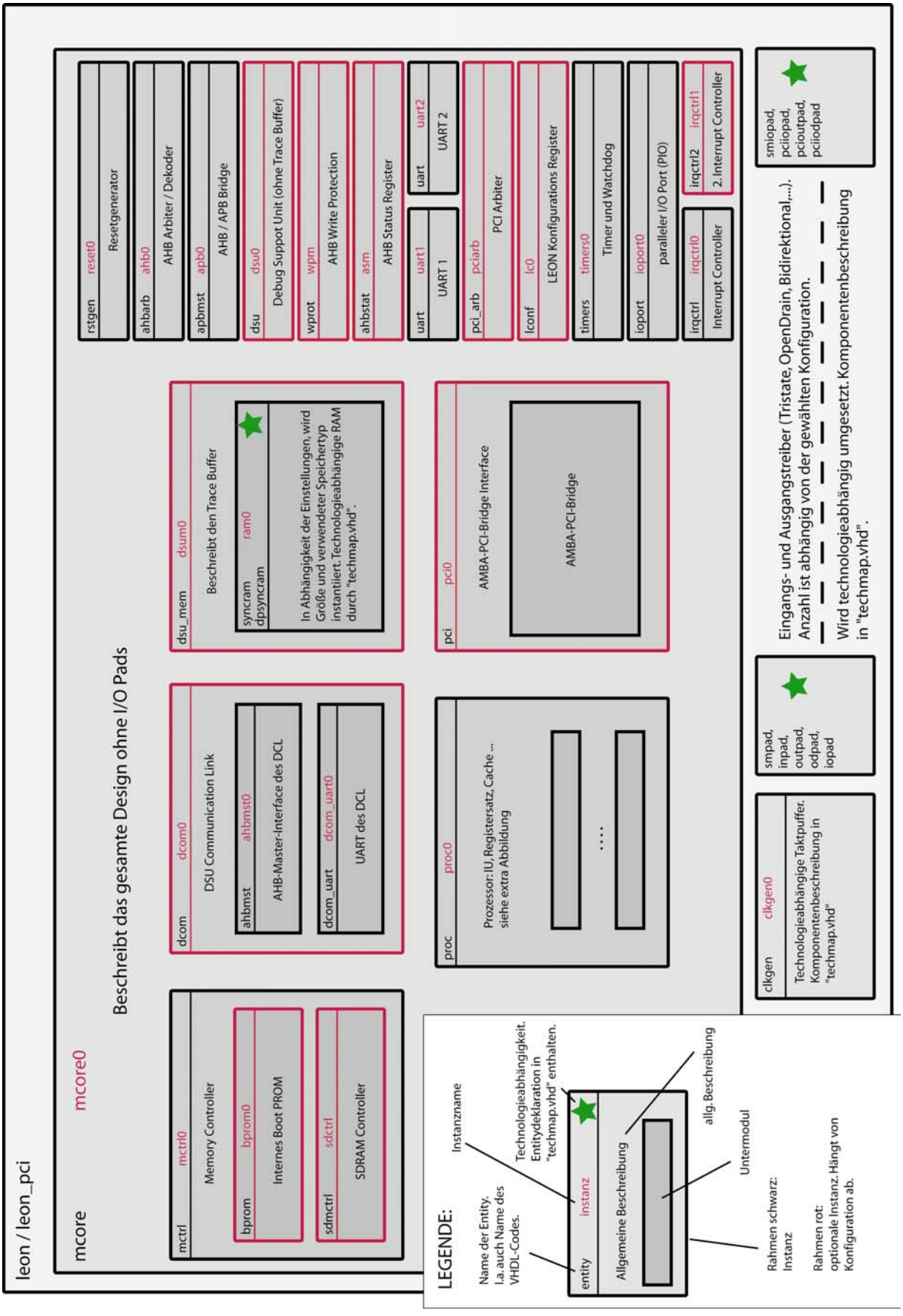


Abb. 2-26 Hierarchie LEON

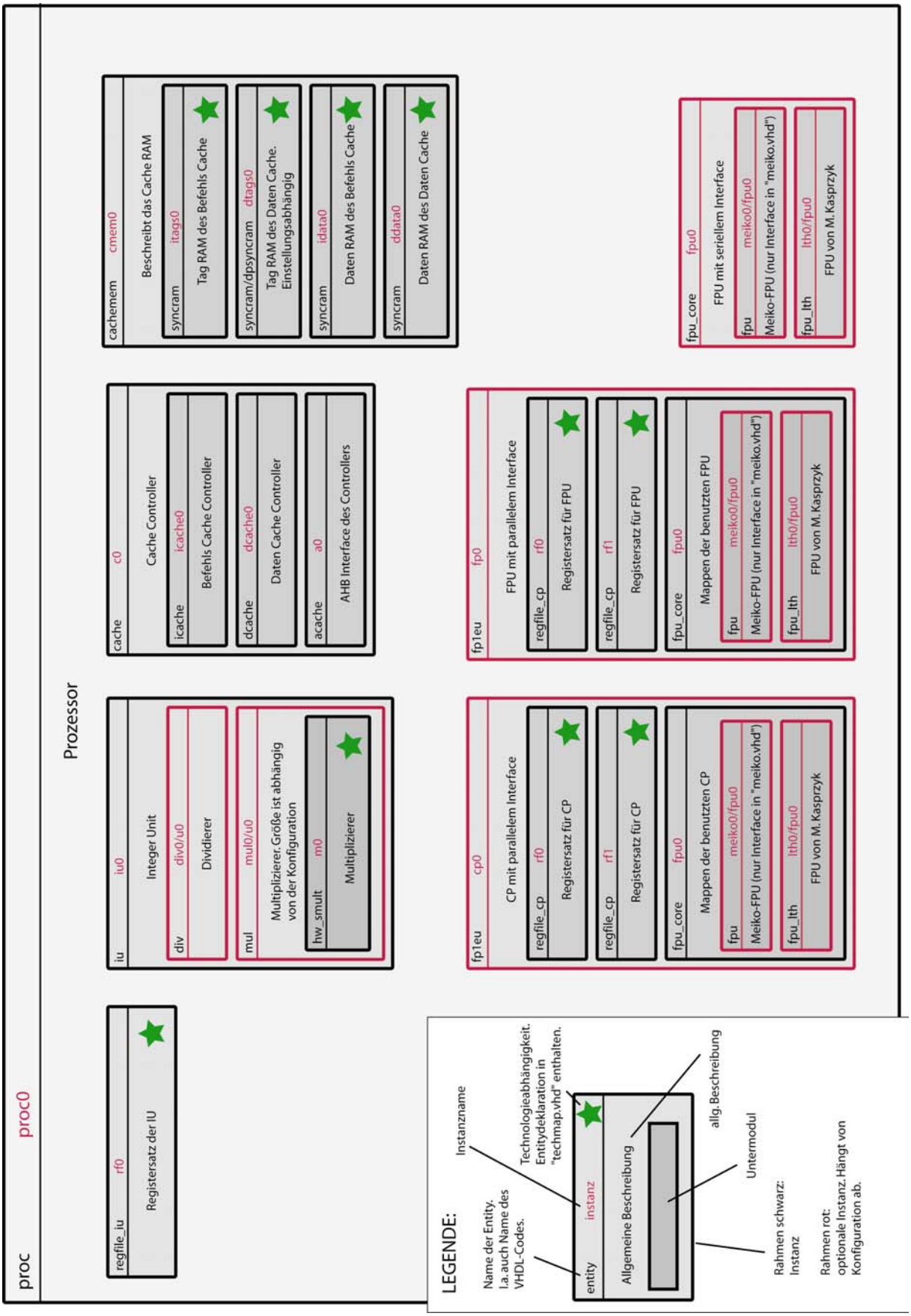


Abb. 2-27 Hierarchie des Prozessors

2.2.2. AMBA-PCI-Bridge

Die Firma Astrium GmbH besitzt einen IP-Core im Umfang von ca. 6000 Zeilen, der eine AMBA-PCI-Bridge beschreibt. Die Aufgabe der AMBA-PCI-Bridge ist, bei Übertragungen zwischen Komponenten, die an verschiedene Bussysteme angeschlossen sind, die Busprotokolle zu übersetzen. Diese Bussysteme sind der AMBA AHB-Bus on-chip und der PCI-Bus extern.

Der verwendete IP-Core unterstützt 32-Bit PCI bei einer Taktfrequenz von 33 MHz. Die in HDL-Code beschriebene Bridge kann auch als Host-Bridge eingesetzt werden, d.h. durch sie können andere Busteilnehmer am PCI-Bus konfiguriert werden. PCI-Interrupts werden nicht verarbeitet.

Um die Bridge vom LEON aus konfigurieren zu können, wird der APB-Bus benutzt. Die Bridge hat deshalb zusätzlich ein APB-Interface. Abb. 2-28 zeigt schematisch den Aufbau der Bridge.

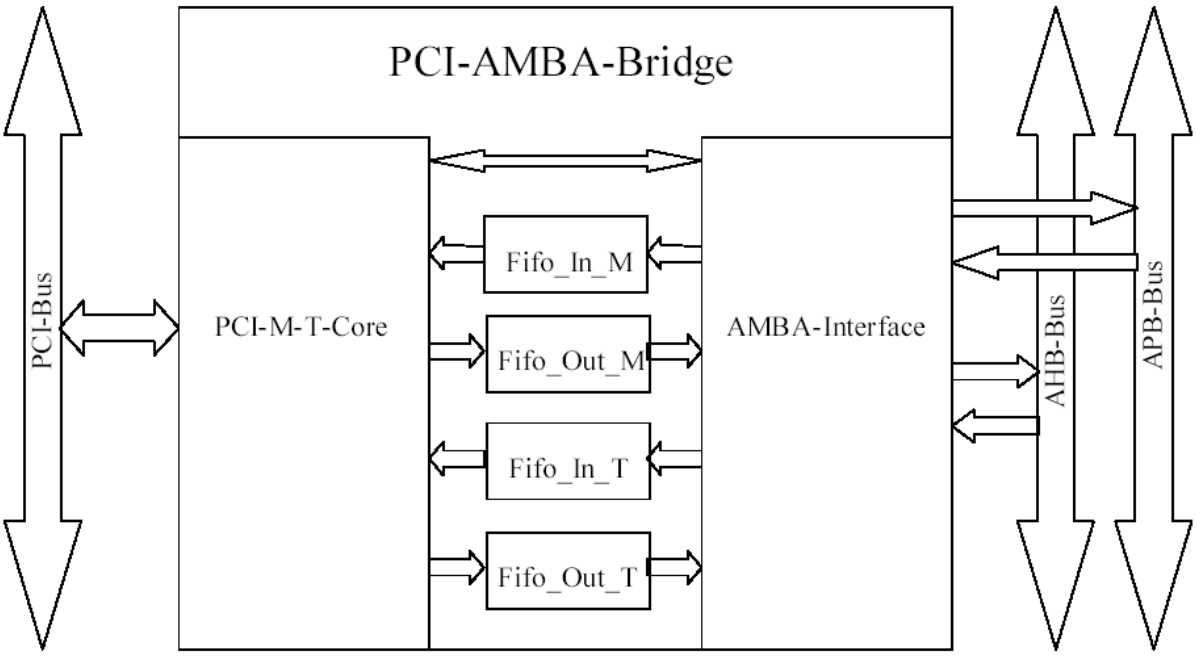


Abb. 2-28 Block Diagramm AMBA-PCI-Bridge (Quelle: [L15])

Der Block PCI-M-T-Core enthält das komplette PCI-Interface mitsamt dem Konfigurationsbereich, der für den PCI-Bus notwendig ist. Der Block AMBA-Interface enthält alle Komponenten, die für die Anbindung an den AHB- und den APB-Bus notwendig sind. Daten werden zwischen diesen beiden Blöcken durch vier FIFOs übertragen. Je zwei sind für eine Übertragungsrichtung zuständig: von AHB-zu-PCI oder PCI-zu-AHB.
Die Konfigurierung des PCI-Interfaces über den APB-Bus geschieht nicht durch diese FIFOs sondern durch spezielle Steuerleitungen.

Adressbereiche einzelner Komponenten, die an einen Bus angeschlossen sind, können entweder fest oder dynamisch sein. Die Adressbereiche des AHB-Busses des LEON sind nach der Synthetisierung fest vorgegeben. Sie können nicht mehr dynamisch im Betrieb des LEONs geändert werden. Auf dem PCI-Bus werden Adressbereiche dynamisch vergeben.

Ist der LEON das Hostsystem, stellt dies kein Problem dar. Die Systemsoftware des LEON kann allen externen, am PCI-Bus angeschlossenen Komponenten die Adressbereiche zuweisen, die noch nicht intern vergeben sind.
Ist des LEON nicht das Hostsystem kann dies u.U. zu Problemen führen, da der Bridge vom Hostsystem ein Adressbereich zugewiesen wird, der nicht dem dahinterliegenden AHB-Adressbereich entspricht.
Um diese Probleme zu umgehen, kann die Bridge während der Synthetisierung so konfiguriert werden, dass Adressbereiche, die auf dem PCI-Bus für die hinter der Bridge liegenden AHB-Komponenten vergeben sind, bridgeintern auf die starren Adressbereiche des AHB-Bus umgewandelt werden.

Auf dem PCI-Bus wird zwischen Speicher-, I/O- und Konfigurationszugriffen unterschieden. Auf dem AHB-Bus gibt es diese Unterscheidung nicht, d.h. die Bridge kann bei Zugriffsrichtung AHB-zu-PCI nicht anhand der AHB-Signale unterscheiden, welche Zugriffsart gemeint ist. Sollte der LEON als Host eingesetzt werden, müssen Konfigurationszugriffe gewährleistet sein, um die PCI-Komponenten zu konfigurieren. Außerdem müssen Zugriffe auf Speicher oder I/O-Bereiche möglich sein, damit die Treiber der Komponenten auf sie zugreifen können. Um dies zu gewährleisten, sind zwei verschiedene Arten der Identifizierung der Zugriffsart durch die Bridge möglich.

Eine Möglichkeit ist, einen Adressbereich auf dem AHB-Bus (innerhalb des Adressbereiches, der der Bridge zugewiesen ist) festzulegen, der für Konfigurations- oder I/O-Zugriffe reserviert ist. Zugriffe auf diesen Bereich werden von der Bridge als Konfigurations- oder I/O-Zugriffe an dem PCI-Bus weitergegeben. Normalerweise sind diese Bereiche nicht so umfangreich wie Speicherbereiche, weshalb sie nicht viel Adressbereich benötigen. Alle anderen Adressbereiche außerhalb werden von der Bridge als Speicherzugriffe interpretiert.

Die andere Möglichkeit besteht darin, spezielle Register zu benutzen, die im AMBA-Interface Block der Bridge implementiert sind. Diese sind über den APB-Bus adressierbar und legen fest, welche Zugriffsart an das PCI-Interface signalisiert werden soll, wenn die Bridge seitens des AHB-Busses adressiert wird. Zusätzlich können Adressbits der Adresse auf dem AHB-Bus für den PCI-Bus durch andere ersetzt werden. Dies kann dazu benutzt werden, um auf Adressbereiche auf dem PCI-Bus zuzugreifen, die AHB intern durch LEON Komponenten vergeben sind.

Ob und welche dieser beiden Arten eingesetzt wird, ist durch Konfigurationseinstellungen zur Synthese festzulegen.

Zu Testzwecken werden anwenderspezifische Register in den Konfigurationsbereich des PCI-Interfaces implementiert. Dieser kann sowohl von PCI- als auch von AHB-Seite aus adressiert werden.

3. Eigenentwicklungen

3.1. PMC-LEON-Board

Der LEON soll mit notwendiger Peripherie als Mezzanine Card zum Einsatz kommen und ein älteres Prozessor-Modell ablösen. Dafür ist als erster Schritt ein Demonstrator zu entwickeln, der getestet werden kann. Dieses Demonstrations-Board ist im Rahmen der Diplomarbeit entwickelt worden. Dabei sind die Ziele und Anforderungen für einen Demonstrator mit denen des angestrebten Endproduktes **nicht** identisch.

Diese Diplomarbeit befasst sich nur mit dem Auswählen der Komponenten und deren Vernetzung. Die Layout-Arbeiten werden an anderer Stelle gemacht. Jedoch gestaltet sich dies als iterativer Prozess, so dass im Verlauf des Layoutens auch Arbeitsschritte im Bereich des Designs durchgeführt werden müssen.

Als Referenz für dieses Design stehen zwei andere Designs zu Verfügung. Das *Fault Managment Element* (FME) der Firma Astrium GmbH ist eine PMC-Karte. Dieses System dient zur Fehlererkennung und -korrektur und kann als Referenz für die PMC-Schnittstelle benutzt werden. Das AVNET Board (s. Kapitel 2.1.1), das auch als Testboard für den LEON verwendet wird, dient als Referenz für Testschalter, LEDs etc. und Treiberbausteine.

Zuerst wird in Kapitel 3.1.1 auf die Anforderungen, die an das Board gestellt werden, eingegangen. Anschließend werden die daraus resultierenden Probleme und Kompromisse thematisiert.
Kapitel 3.1.3 beschäftigt sich mit den verwendeten Komponenten. Es wird begründet, warum diese Komponenten verwendet werden, und wenn nötig, eine kurze Beschreibung der Funktionalität gegeben. Ob die Bauteile für die Weltraumfahrt geforderte Qualifikationen erfüllen, wird hier auch betrachtet.
Kapitel 3.1.4 beschreibt die Überlegungen, die zum entwickelten Design geführt haben und das Design selber. Das Kapitel ist in die einzelnen funktionalen Module gegliedert, die jene im Anhang vorhandenen Schematics repräsentieren.

3.1.1. Anforderungsliste

Seitens der Firma Astrium GmbH sind bestimmte Anforderungen an das zu entwickelnde Board gestellt. Diese sind im Folgenden aufgelistet. Dabei ist in Anforderungen oder Vorgaben bezüglich Funktionalität, Schnittstellen und allgemeine Anforderungen unterteilt.

Funktionalität

- Auf dem Demonstrationsboard muss in einem FPGA das LEON-System inklusive PCI-Bridge und (als IP-Core vorhandenem) Ethernet MAC (*Media Access Control*) implementiert werden können. Der Ethernet MAC dient als Bindeglied zwischen Software und dem physikalischen Layer (PHY).
- Das Board muss „*stand alone*“ und als Mezzanine Card betrieben werden können.
- Der auf dem Board vorhandene Speicher (RAM und ROM) soll möglichst groß sein.

- Der FPGA muss im MasterSerial Modus und im SlectMap Modus konfigurierbar sein. Desweiteren muss Scrubbing(s. Kapitel 3.1.3.3) möglich sein.
- Das Board muss mit 3.3V und 5V 32-Bit-PCI-Bus funktionieren. Zur Generierung der verschiedenen Versorgungsspannungen soll die 5 V Leitung des PCI-Busses benutzt werden.
- Es sollen LEDs, Taster und Schalter für Testzwecke vorhanden sein.
- Es müssen gleichzeitig beide UARTs und der DSU Communication Link des LEON benutzbar sein. Es muss der RS232 Standard zur Übertragung benutzt werden.
- In System Programming (ISP) über JTAG muss möglich sein.

Schnittstellen

- PMC-Schnittstelle: 32-Bit PCI + User I/O
- Front Bezel : 1*DSUB9 für RS232, 1* RJ45-Buchse (8 polig) und LEDs für Ethernet
- Abgriffe für Masse und Versorgungsspannungen für Messungen bzw. externe Spannungsversorgung
- Zusätzlicher Steckverbinder für Flachbandkabel (für JTAG und restliche UARTs), das zu externer Leiterplatte führt.

Allgemeine Anforderungen

- Das Endprodukt soll die Spezifikationen für Flug- und Weltraumtauglichkeit erfüllen. Um den Aufwand eines Redesigns zu minimieren, soll das Demonstrations-Board so weit wie möglich nach diesen Richtlinien entwickelt werden.
- Die PMC-Spezifikation und CMC-Spezifikation soll eingehalten werden.
- Finanzielle Aspekte

3.1.2. Problematiken

Gewisse Anforderungen sind miteinander nicht in Einklang zubringen. Im Folgenden wird das näher beleuchtet und anschließend werden die Ergebnisse -also die Kompromisse daraus - betrachtet.

3.1.2.1. Qualifizierte Bauteile

Bauteile, die für den Einsatz im Weltraum qualifiziert sind, erfüllen bestimmte Anforderungen. Auf der einen Seite betrifft dies mechanische und thermale Belastbarkeit, auf der anderen Seite Fehlersicherheit gegenüber Strahlung, die im Weltraum herrscht. Die Weltraumstrahlung setzt sich aus dem Sonnenwind und kosmischer Strahlung zusammen und besteht aus Protonen und Ionen. Beide Teilchen sind so hochenergetisch, dass sie auch feste Materie durchschlagen können, also auch Plastikhüllen eines FPGAs oder Speicherbausteins. Beim Auftreffen der Teilchen auf Hardwarestrukturen, wie z.B. Flipflops oder Speicherzellen, können sie Fehler hervorrufen.
Strahlungsfeste Versionen gibt es nicht für alle Bauteiltypen und sie sind auch in aller Regel teurer als Standard-Bauteile.

3.1.2.2. Zusätzliche Bauteile für Demonstrationszwecke und Konfiguration

Zu Testzwecken ist es sinnvoll, visuelle Signale, Taster und Schalter zu Verfügung zu haben. Auch Abgriffmöglichkeiten für Masse und Spannungen sind zum Messen von Signalen nötig. Wie weiter oben erwähnt, sollen zwei verschiedene Arten der FPGA-Konfiguration möglich sein.
All dies benötigt Bauteile, die in einer kommerziellen Version nicht vorhanden sein müssen. Durch die zusätzlichen Bauteile wird mehr Platz auf der durch die PMC-Spezifikation vorgegebenen Fläche benötigt.

3.1.2.3. PMC-Spezifikation

Unter 2.1.3 wurde die PMC bzw. CMC-Spezifikation zusammengefasst. Diese gibt eine Einhüllende für alle Bauteilkomponenten vor. Nur innerhalb dieser Einhüllenden dürfen sich Bauteile befinden. Besonders auf Seite 2 der PMC-Karte und im Front Bezel Bereich stellt dies hohe Anforderungen bezüglich der räumlichen Ausdehnung der verwendeten Komponenten. So ist keine RJ45-Buchse zu bekommen, die nur 11mm hoch ist. Dies ist aber die maximal zulässige Höhe.
Jumper oder Pfostenstecker müssen zugänglich angebracht sein. Das bedeutet, dass sie auf der Seite 2 der PMC-Karte gemountet sein müssen. Dort ist aber nur Platz für ca. 1,7mm hohe Bauteile (bei einer veranschlagten PCB-Dicke von 1,8 mm). Dies ist mit den genannten Komponenten nicht zu erreichen, ohne Komfortabilität in der Benutzung des Boards zu verlieren.

3.1.2.4. Richtlinien

Eine Abwägung der in den vorherigen Kapiteln beschriebenen Gesichtspunkte führte zu folgenden Richtlinien:

- Damit das entwickelte Board mit möglichst wenig Aufwand für ein kommerzielles Produkt redesigned werden kann, sollen möglichst viele Bauteile entweder strahlungsfest sein oder als strahlungsfestes Äquivalent existieren. Dies bedeutet, dass die Lötmaske für ein Bauteil und das zugrundeliegende Pinout identisch sind.
- Im Hinblick auf Wärmeerzeugung und Stromversorgung wird nicht auf Flugtauglichkeit hin entwickelt. Dies setzt Analysen und daraus resultierende Lösungskonzepte voraus, die in einem Redesign gemacht werden.
- Bauteile, die nur für das Demonstrationsboard verwendet werden und zugänglich sein müssen, sind auf Seite 2 des Boards zu platzieren. Eventuelle Verletzungen der PMC-Spezifikation sind vernachlässigbar, da die Komponenten im Redesign entfernt würden. Dies betrifft Jumper, Schalter und Taster.
- Bezüglich der Höhe von Steckern im Front Bezel Bereich wird eine Verletzung der PMC-Spezifikation zugelassen, um Standard-Komponenten verwenden zu können. Dies wird in einer geringen Schieflage der PMC-Karte resultieren. Andere PMC-Testkarten von Astrium, bei denen auch eine Überhöhung dieses I/O-Bereiches vorliegt, zeigen, dass die Kontaktierung der PMC-Steckverbinder gewährleistet bleibt.
- Um die Kosten möglichst niedrig zu halten, sollen bevorzugt Bauteile verwendet werden, die auf Lager liegen.

3.1.3. Bauteile

Anhand der Anforderungen und Richtlinien, den verwendeten Bauteilen in den Referenzdesigns und in Absprache mit Lutz Buttelmann und dem Einkäufer von Bauteilen sind die Komponenten ausgewählt worden.

3.1.3.1. FPGA: XCV1000 –5 BG560 C

Astrium GmbH benutzt bisher strahlungsfeste FPGAs der Firma Actel. Aufgrund der höheren Integrationsdichte der FPGAs der Firma Xilinx (siehe [L4]), ist das Interesse groß, diese in zukünftigen Produkten zu verwenden. Nach [L4] stellt die Verwendung der Xilinx-Bausteine auch einen Kostenvorteil gegenüber denen der Firma Actel dar. Das Evaluations-Board der Firma AVNET ist mit einem Virtex E bestückt. Unter Berücksichtigung dieser drei Gesichtspunkte wird die Wahl eines Xilinx Bausteines bevorzugt. Die Gründe für die Wahl des XCV1000 –5 BG560 C sind im Folgenden aufgeführt.

- Der FPGA muss neben dem LEON auch die AMBA-PCI-Bridge(Kapitel 2.2.2) und den in Kapitel 3.1.1 erwähnten IP-Core eines Ethernet MAC enthalten können. Der Platzbedarf für LEON, AMBA-PCI-Bridge und Ethernet MAC wird durch Durchlaufen des Place&Route Prozesses für einen XCV1000E ermittelt. Dieser FPGA kommt auf dem Evaluations-Board von AVNET zum Einsatz. Es werden durch das Design verwendet:

  LEON: 5148 Slices / 24 BlockRAMs
  Bridge: 1217 Slices
  Ethernet MAC: 1850 Slices

  Dies ergibt insgesamt einen Platzbedarf von 8215 Slices. Die Anzahl der benutzten BlockRAMs ist abhängig von der Konfiguration des LEONs. Mit einer Sicherheits-Margin von ca. 50% muss der verwendete FPGA ca. 12000 Slices enthalten. Xilinx FPGAs beinhalten pro CLB 2 Slices. Anhand der Datenblätter zeigt sich, dass mindestens ein Virtex 1000 oder VirtexE 1000 Baustein eingesetzt werden muss, um dieses Kriterium zu erfüllen. Der Virtex 1000 besitzt 12288 Slices und 32 BlockRAMs, während der VirtexE 1000 bei gleicher Anzahl Slices 96 BlockRAMs zu Verfügung stellt.

- Die Virtex-Reihe ist im Gegensatz zur Virtex E-Reihe 5V-Tolerant, ohne zusätzliche externe Beschaltung. Als strahlungstolerante Version gibt es nur solche aus der Virtex Reihe.

- Anhand der Entities von LEON, Bridge und VMAC lässt sich die Anzahl von Ports ermitteln, die unabhängig von eventuellen zusätzlichen, vom Design abhängigen Ports benutzt werden.

  LEON: 120 Ports
  Bridge: 65 Ports
  VMAC: 27 Ports

  Insgesamt sollten mindestens 212 I/O-Ports durch den FPGA bereitgestellt werden. Für einen Virtex 1000 kommen nur zwei verschiedene Packages in Frage:

BG560 und FG680: Beides sind Ball Grid Arrays, doch nur für das Package BG560 gibt es eine strahlungstolerante Version. Mit 404 I/O-Ports bietet das Package genug Platz für zusätzliche, noch nicht verplante Ports.

- Es stehen die Geschwindigkeitsklassen -4, -5, -6 für die Virtex-Reihe zu Verfügung. Dabei garantiert die Klasse -6 schnellere Verarbeitungs-Geschwindigkeiten als Klasse -5, Klasse -5 schnellere als Klasse -4. Wünschenswert ist demnach die Geschwindigkeitsklasse -6. Es zeigt sich jedoch, dass es einen unverhältnismäßig hohen Preisanstieg zwischen den Klassen -5 und -6 gibt. Deshalb wird ein Virtex der Geschwindigkeitsklasse -5 benutzt.

- Für das Demonstrations-Board reicht die kommerzielle Variante des Virtex 1000 aus (Index C).

3.1.3.2. Speichertypen: AM29F016D und uPD434008ALE -15

Als strahlungsfester Speicher ist nur SRAM qualifiziert, mit einer maximalen Größe von 512k*8Bit. Das Gehäuse gibt es im SOJ 36 Package. Es wird ein äquivalentes, kommerzielles Bauteil verwendet. Dies ist der Baustein uPD43008ALE-15 der Firma NEC mit einer maximalen Zugriffszeit von 15 ns. Es werden insgesamt 8 Bausteine verwendet mit einer Gesamtspeicherkapazität von 4MB.

Als PROM wird Flash-Speicher verwendet, um eine Wiederbeschreibbarkeit zu gewährleisten. In anderen Boards der Firma Astrium GmbH wird schon ein Typ verwendet, der zwar keine guten Strahlungswerte besitzt, aber trotzdem toleriert wird. Dies ist der Baustein AM29F016D der Firma AMD im Package TSOP48. Er hat eine Speicherkapazität von 2MB in 8-Bit Organisation. Es werden 4 Bausteine davon als PROM des LEONs benutzt und einer als Konfigurationsflash.

3.1.3.3. FPGA-Konfiguration

Eine direkte Programmierung des FPGA über die JTAG-Schnittstelle benötigt keine besondere Hardware. Diese Art der Programmierung ist jedoch nicht permanent. Bei Abbruch der Spannungsversorgung verliert der FPGA seine Konfiguration und damit seine Funktionalität. Um eine automatische Konfiguration bei Anlegen einer Spannungsversorgung zu ermöglichen, wird permanenter Speicher benötigt, um die Konfigurationsdaten zu speichern.

Die einfachste Möglichkeit einen FPGA durch einen permanenten Speicher zu konfigurieren, ist der MasterSerial Modus. Dazu benötigt man einen PROM-Speicher, der die Daten seriell an den FPGA während der Konfiguration sendet. Dieses PROM muss beschrieben werden können. Z.B. kann dies über eine serielle Schnittstelle und ein dazwischenliegendes CPLD geschehen.

Xilinx bietet zur Konfiguration seiner FPGAs und CPLDs spezielle PROMs an (XC18V00-Reihe), die über die JTAG-Schnittstelle programmiert werden können, ohne dass ein CPLD benötigt wird. Abb. 3-1 zeigt, dass das CPLD nur deshalb nicht benötigt wird, weil Ansteuerlogik und Seriell/Parallel-Wandlung in die PROMs integriert worden sind. Zusätzlich zum MasterSerial Modus können sie auch ein FPGA im SelectMap Modus

konfigurieren, was den Anforderungen an das Board entspricht. Die Verwendung dieser PROMs vereinfacht die Inbetriebnahme des Boards erheblich, da kein spezielles Design für ein CPLD entwickelt werden muss, um einfache Tests durchzuführen.

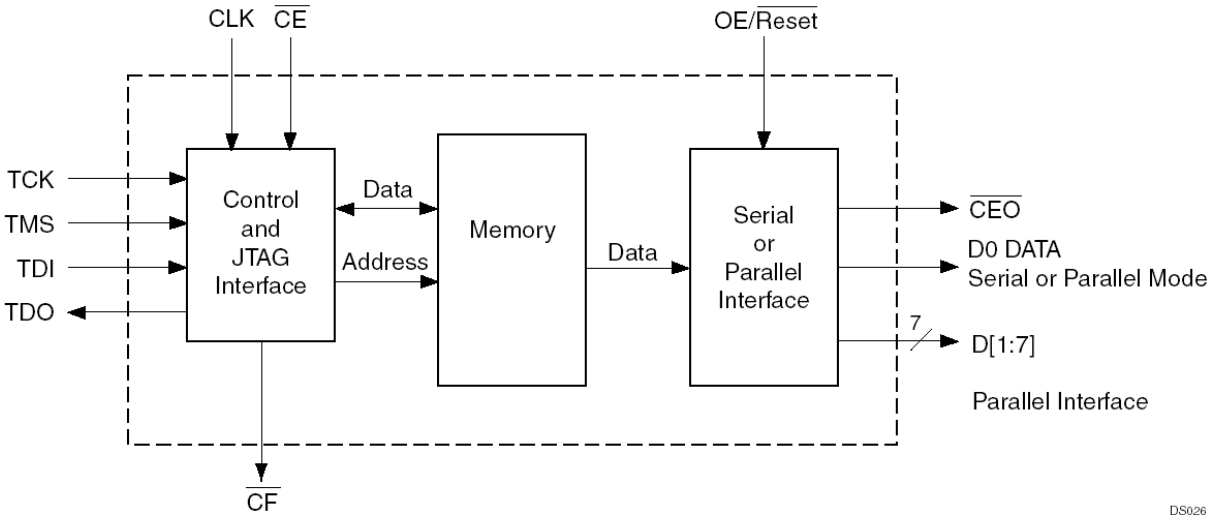


Abb. 3-1 Blockschaltbild Xilinx XC18V00 Serie (Quelle: [D11])

In der XC18V00-Reihe gibt es unterschiedliche PROMs. Sie unterscheiden sich in Speicherplatz und Gehäuse. Die Dokumentation der XC18V00-Bausteine stellt Informationen über die benötigten PROMs für unterschiedliche FPGAs zu Verfügung. So wird für einen XCV1000 eine Kombination aus einem XC18V02 und einem XC18V04 benötigt. Zusammen haben sie eine Speicherkapazität von 6127744 Bits. Dies ist genau die Anzahl, die für einen XCV1000 benötigt wird. Die beiden PROMs müssen kaskadiert werden (s. Kapitel 3.1.4.5). Es kann für diese beiden Typen zwischen zwei verschiedenen Packages gewählt werden: VQ44 und PC44. Als strahlungsfeste Version gibt es nur ein Äquivalent zum XC18V04 VQ44. Da das Layout eines XC18V02 und XC18V04 nur vom Package abhängig ist, können für das Demonstrationsboard beide Typen verwendet werden, ohne dass im Redesign das Layout verändert werden muss. Es wird von beiden Typen die kommerzielle Version benutzt: XC18V02VQ44C und XC18V04VQ44C.

Als Scrubbing bezeichnet man Konfigurationen im laufenden Betrieb. Es ist eine Methode, um Systeme fehlertoleranter zu machen. Dabei wird der FPGA zyklisch neu konfiguriert (vergl. [L4]). Bei Verwendung allein der Xilinx-PROMs ist die Forderung nach der Möglichkeit von Scrubbing nicht erfüllt, da beim Scrubbing bestimmte Teile des Bit-Streams, mit dem der FPGA initial konfiguriert wird, nicht verwendet werden. Dies betrifft z.B. die Inhalte von BlockRAMs. Um diese Teile bei der zyklischen Rekonfiguration zu unterdrücken, ist zwangsläufig ein CPLD oder eine andere Steuerlogik notwendig, da die Xilinx PROMs diese Funktionalität nicht implementiert haben.

Die Tests, die in [L4] bezüglich Scrubbing beschrieben werden, verwenden ein CPLD der Firma Xilinx (XC95108). Dieses hat 108 Macrozellen und 81 benutzbare I/O-Ports. In Zukunft sollen die Erkenntnisse aus [L4] weiterverwendet werden. Deshalb wird ein CPLD benutzt. Die Funktionalität des CPLDs ist nicht Thema dieser Diplomarbeit und wird deshalb nicht weiter beschrieben.
Seitens Astrium ist der XC95144XL vorgegeben. Dieser hat 144 Macrozellen. Es wird das Package TQFP144 benutzt. Dadurch stehen 117 benutzbare I/O-Ports zu Verfügung. Dies soll Flexibilität und zusätzliche noch nicht funktional belegte Verbindungen zwischen FPGA und CPLD gewährleisten. Es wird die kommerzielle Variante benutzt. Ein qualifiziertes

Äquivalent ist nicht bekannt. Die Produktbezeichnung ist abweichend vom Package Typ XC95144XL –7 PC44 C.

Scrubbing muss auch auf einen Speicher zugreifen, in dem die Konfigurationsdaten abgelegt sind. Die PROMs der Firma Xilinx sind nicht direkt adressierbar. Um die Scrubbing Methode entsprechend [L4] einzusetzen, muss ein parallel adressierbares PROM vorhanden sein. Dazu wird das in Kapitel 3.1.3.2 beschriebene Flash-PROM AM29F016D benutzt. Mit 16 Mbit Speicherkapazität ist es mehr als ausreichend.

3.1.3.4. Schnittstellen Komponenten

Als Treiber für die RS232 Schnittstellen wird ein 5-fach Transceiver der Firma Linear Technology verwendet (LT1130A). Dieser hat 5 Sende- und 5 Empfangsleitungen. Er wandelt beim Senden TTL/CMOS-Signale in RS232-Pegel und beim Empfangen RS232-Pegel in TTL/CMOS-Signale um. Als RS232-Spannungspegel generiert sich die Komponente aus der 5V Versorgungsspannung nach Herstellerangaben typischerweise +7.3V und –6.5V. Es wird die kommerzielle Version im SW-Package (SO28)benutzt: LT1130ACSW.

Als Treiber für die Ethernet Schnittstelle wird ein Baustein benötigt, der die untersten physikalischen Layer implementiert hat. Diese Bausteine werden unter dem Begriff Ethernet PHY geführt. Da keine strahlungstolerante Version bekannt ist, wird die Komponente verwendet, die im Referenzdesign von AVNET verwendet wird. Dies ist der Baustein LXT971A der Firma Intel. Er unterstützt 10 Mbps und 100 Mbps Netzwerke im Halb- und Vollduplexbetrieb. Als Kontroll-Interface für einen Ethernet MAC stellt er das Media Independent Interface (MII) zu Verfügung. Der Baustein wird als 64-Pin LQFP Gehäuse gewählt.

3.1.3.5. Spannungsversorgung

Es werden 3 verschiedene Versorgungsspannungen für Bauteile auf dem Board benötigt: 5V, 3,3V und 2,5V. Da vorgegeben ist, alle Spannungen aus der 5V–Leitung des PCI-Bus zu generieren, kann die 3,3V Leitung desselben nur optional benutzt werden. Funktionsfähigkeit bei ausschließlicher Nutzung der 5V Leitung muss möglich sein.

Es stehen zwei Möglichkeiten zu Verfügung, aus der 5V Spannung die niedrigeren Spannungen zu generieren. Man benutzt entweder Linearregler oder Schaltnetzregler. Der Vorteil der Schaltnetzregler ist eine geringere Verlustleistung und damit weniger Temperaturentwicklung. Für raumfahrttechnische Anwendungen sind diese beiden Aspekte wichtige Kriterien.
Linearregler sind deutlich kleiner als Schaltnetzregler. Wie in Kapitel 3.1.2.4 beschrieben, ist sich darauf geeinigt worden, Überlegungen zur Flugtauglichkeit bei der Stromversorgung in vollem Umfang im Redesign zu machen. Deshalb werden Linearregler verwendet. Es sind die SMD-Linearregler LT1085CM der Firma Linear Technology ausgesucht worden. Die erzeugte Ausgangsspannung ist abhängig von der Beschaltung mit einem Widerstandteiler, deshalb wird für beide Spannungen der gleiche Komponententyp verwendet.

Sinkt die Versorgungsspannung ab, ist die Funktionsfähigkeit der Komponenten nicht mehr gewährleistet. Allerdings ist der Spannungsabfall, ab dem ein Bauteil nicht mehr funktioniert, von Bauteil zu Bauteil unterschiedlich. Um zu vermeiden, dass ein Bauteil ausfällt, während die anderen noch funktionieren, und damit undefinierte Systemzustände zu produzieren, wird

eine gesonderte Komponente zur Spannungsüberwachung verwendet. Diese erzeugt bei Unterschreitung einer bestimmten Spannung ein Reset-Signal. Dies gewährleistet, dass nach Spannungszusammenbruch und anschließender Regeneration die Komponenten in einen definierten Zustand gebracht werden. Zusätzlich dazu hat die verwendete Komponente einen Eingang, über den manuell ein Reset ausgelöst werden kann. Diese Funktionalität wird zwar nicht verlangt, scheint jedoch als Option sinnvoll.
Der verwendete Baustein ist der MAX706T der Firma Maxim. Dieser überwacht eine 3,3V Spannung. Sinkt sie unter 3.08V, wird ein Reset ausgelöst. Für das entwickelte Design wäre eine 5V Spannungsüberwachung angebrachter (z.B. MAX706), weil verschiedene Komponenten die 5V als Betriebsspannung verwenden. Jedoch liegt der MAX706T auf Lager und soll deshalb verwendet werden. Er liegt im SO 8 Package vor.

3.1.3.6. Diskrete Bauteile

Als diskrete Bauteile werden Widerstände und Kondensatoren verwendet. Diese werden als SMD Version benutzt, um Platz zu sparen. Als Trough Hole Version werden 3 LEDs verwendet, die am Front Bezel der PMC-Karte den Status der Ethernetschnittstelle anzeigen sollen. Auch die RJ45-Buchse und der DSUB9-Stecker sind Through Hole Versionen, um mechanisch belastbarer zu sein.

Zwei RS232 Schnittstellen, das JTAG Interface sowie Spannungsleitungen werden über einen SMD-Flachbandkabelstecker FTSH der Firma Samtec vom Board geführt.

Es werden zwei verschiedene SMD-Oszillatoren eingesetzt: Ein 25 MHz Oszillator für den Ethernet PHY (O25VX3MH der Firma Jauch Quartz GmbH), sowie ein 40 MHz Oszillator zur Konfiguration des FPGAs im SlectMap Modus (SG636PTF40MHz der Firma Epson). Beide Taktsignale können als Systemtakt dem FPGA zu Verfügung gestellt werden.

Zu Testzwecken werden 8 Schalter, 4 Taster und 8 LEDs in SMD Version benutzt. Außerdem kommt eine Reihe von Jumpern zum Einsatz. Auf die genaue Bedeutung wird im nächsten Kapitel eingegangen.

3.1.4. Schematics und Funktionalität

Das Design ist nach dem Top-Down-Verfahren entwickelt worden. D. h., dass die Aufgabe von der obersten Hierarchie bzw. höchsten Abstraktionsebene in Funktionseinheiten zergliedert ist, die ihrerseits verfeinert werden, wenn dies nötig erscheint. Dabei sind die Funktionseinheiten weitgehend in sich abgeschlossen, so dass innerhalb ihrer gearbeitet werden kann, ohne andere Funktionseinheiten verändern zu müssen. Der Top-Level heißt PMC_LEON_TOP.
Für die gesetzte Aufgabe wurden folgende Funktionseinheiten gewählt, die sich in der obersten Hierarchie befinden (s. Anhang A-2 Blatt: 1/13 Name: pmc_leon_top):

- VIRTEX1000: Stellt das FPGA dar.
- CONNECTOR_BLOCK: Flachbandkabel-Stecker.
- POW_RES_CLK: Die Generierung der Spannungen, die Überwachung derselben und die Erzeugung der Taktsignale wird hier beschrieben.
- JTAG_IF_CFG_PROM: Enthält alle speziell zur Konfiguration benötigten Bauteile.
- SRAM: Dient dem LEON als SRAM.

- ROM: Dient dem LEON als ROM.
- UART_IF: Beschreibt die Verdrahtung des Treiberbausteins der UARTs und den DSUB9 Stecker.
- LED_SW_BUT: Enthält die LEDs, Schalter und Taster, die zu Testzwecken implementiert werden.
- ETHERNET_IF: Beschreibt die Verdrahtung des Ethernet PHYs, der Status-LEDs und der RJ45-Buchse.
- PMC_IF: Stellt die PMC-Steckverbinder dar. Spannungspufferung.

Die Zergliederung in zwei Hierarchieebenen ist für das Design ausreichend. Die einzelnen Hierarchien werden in den nächsten Unterpunkten beschrieben. Dabei wird sich auf die im Anhang vorhandenen Schematics bezogen (A-2).

Sowohl für FPGA als auch für das CPLD muss ein Pinout festgelegt werden. Dies ist aufgrund von Platzierungen so gemacht worden, dass Leiterbahnen sich möglichst nicht überkreuzen.

3.1.4.1. PMC_TOP_LEVEL

(s. A-2: Blatt 1/13 und 2/13)
SEITE 1

Seite 1 des Top-Level Designs enthält die unter Kapitel 3.1.4 beschriebenen Blöcke und ihre Verbindungen untereinander. In Tabelle 3-1 sind die einzelnen Funktionsgruppen dargestellt. Die Signalleitungen haben keine Namen, deshalb wird sich auf die Port Namen der Blöcke bezogen. Mehrere Blöcke können für ein Signal unterschiedliche Portnamen haben. Die Notation für die Portnamen berücksichtigt dies:

Name1[A,A,...,A](*Name 2*[B,..,B])(*Name 3*[C,...,C])....

Name 1: Portname von Block Nr.1 für ein Signal
Name 2: von *Name 1* abweichender Portname für das entsprechende Signal
Name 3: von *Name 1* und *Name 2* abweichender Portname
A: Blocknummern aller an ein Signal angeschlossenen Blöcke
B: Blocknummern, deren Portname *Name 2* für das entsprechende Signal ist
C: Blocknummern, deren Portname *Name 3* für das entsprechende Signal ist

Die Tabelle beschreibt die Signalfunktionen. Deshalb wird jedes Signal nur einmal aufgeführt. D.h., dass Portnamen nicht doppelt aufgeführt werden.

Name[Blöcke](opt. Name[Block])	Blöcke	Funktion
CON_UART2_TX[1,2], CON_UART2_RX[1,2], CON_UART2_CTS[1,2], CON_UART2_RTS[1,2]	1)CONNECTOR_BLOCK 2)UART_IF	Eine RS232 Schnittstelle mit Handshake-Möglichkeit. (1) enthält die Steckverbindung, (2) den Treiber.
CON_UART3_TX[1,2], CON_UART3_RX[1,2]	1)CONNECTOR_BLOCK 2)UART_IF	Eine RS232 Schnittstelle ohne Handshake-Möglichkeit. (1) enthält die Steckverbindung, (2) den Treiber.
CON_JTAG_TDO[1,3](TDO[3]), CON_JTAG_TMS[1,2,3](TMS[2,3]), CON_JTAG_TCK[1,2,3](TCK[2,3]), CON_JTAG_TDI[1,2](TDI[2])	1)CONNECTOR_BLOCK, 2)JTAG_IF_CFG_PROM, 3)VIRTEX1000	JTAG-Interface. Wird an (1) angeschlossen. Reihenfolge: erst (2), dann (3).

CON_JTAG_CCLK[1,2,3] (CCLK[2,3]), CON_JTAG_PROG_N[1,2,3,4,5] (PROG_N[2,3]) (POW_RESET_N[4]) (RESET_N[5]), CON_JTAG_INIT_N[1,2,3] (INIT_N[2,3]), CON_JTAG_DONE[1,2,3] (DONE[2,3]), CON_JTAG_BUSY/DOUT[1,2,3] (BUSY/DOUT[2,3]), CON_JTAG_CS_N[1,2,3] (CS_N[2,3]), CON_JTAG_WRITE_N[1,2,3] (WRITE_N[2,3]), CON_JTAG_DATA[1,2,3] (PROG_DATA[2,3]),	1)CONNECTOR_BLOCK, 2)JTAG_IF_CFG_PROM, 3)VIRTEX1000, 4)POW_RES_CLK, 5)ROM	Konfigurationssignale für Virtex. Bei einem Reset durch (4), wird neu konfiguriert und das FLASH-PROM in (5) in Lesezustand versetzt.
CLK_40MHZ[1,2]	1)POW_RES_CLK, 2)JTAG_IF_CFG_PROM	40 MHz Takt, der für SelectMap Modus verwendet wird als CCLK.
CLK_25MHZ[1,2] (ENET_25MHZ[2])	1)POW_RES_CLK, 2)ETHERNET_IF	25 MHz Takt für Ethernet PHY.
CLK[1,2]	1)POW_RES_CLK, 2)VIRTEX1000	Systemtakt für LEON. Entweder 25 oder 40 MHz.
MR[1,2]	1)POW_RES_CLK, 2)JTAG_IF_CFG_PROM	Manueller Reset. Wenn aktiv wird POW_RESET_N von (1) aktiviert, und damit Konfiguration des Virtex neu gestartet.
WDI[1,2], WDO[1,2]	1)POW_RES_CLK, 2)JTAG_IF_CFG_PROM	WatchdogIn, WatchdogOut der Spannungsüberwachung. Für zukünftige Designs vorgesehen.
MODE0[1,2], MODE1[1,2], MODE2[1,2]	1)JTAG_IF_CFG_PROM, 2)VIRTEX1000	Stellen den Konfigurationsmodus des FPGAs ein.
FPGA_CPLD[1,2]	1)JTAG_IF_CFG_PROM, 2)VIRTEX1000	Nicht funktional benutzte Pins des CPLD sind für zukünftige Designs an den Virtex gelegt.
ADDRESS[1,2,3], DATA[1,2,3],	1)SRAM, 2)ROM, 3)VIRTEX1000	Adress- und Datenleitungen des Speicherbusses des LEON.
RAMS_N[1,2], RAMOE_N[1,2], RWE_N[1,2](RAMWE_N[2])	1)SRAM, 2)VIRTEX1000	Steuersignale für SRAM.
ROMS_N[1,2], ROMOE_N[1,2], WE_N[1,2](ROMWE_N[2], RY/BY[1,2](ROM_RY/BY[2])	1)ROM, 2)VIRTEX1000	Steuersignale für FLASH PROM. RY/BY kann beim Programmieren des Flashs benutzt werden.
UART1_TX[1,2], UART1_RX[1,2], UART1_RTS[1,2], UART1_CTS[1,2],	1)UART_IF, 2)VIRTEX1000	UART mit Handshake-Möglichkeit. (1) enthält RS232 Treiber und DSUB9-Stecker, (2) die UART.
UART2_TX[1,2], UART2_RX[1,2], UART2_RTS[1,2], UART2_CTS[1,2],	1)UART_IF, 2)VIRTEX1000	UART mit Handshake-Möglichkeit. (1) enthält RS232 Treiber, (2) die UART.
UART3_TX[1,2], UART3_RX[1,2],	1)UART_IF, 2)VIRTEX1000	UART ohne Handshake-Möglichkeit. (1) enthält RS232 Treiber, (2) die UART.
LED[1,2], SW[1,2], BT[1,2]	1)LED_SW_BUT, 2)VIRTEX1000	Signalleitungen für LEDs, Taster und Schalter, die zu Testzwecken implementiert werden.

ENET_MDIO[1,2], ENET_MDC[1,2], ENET_MDINT_N[1,2], ENET_MDDIS[1,2],	1)ETHERNET_IF, 2)VIRTEX1000	Media Independent Interface (MII) Signale.
ENET_TX_CLK[1,2], ENET_TX_EN[1,2], ENET_TX_ER[1,2], ENET_TXD[1,2],	1)ETHERNET_IF, 2)VIRTEX1000	Signale zwischen Ethernet MAC und Ethernet PHY für Senden.
ENET_RX_CLK[1,2], ENET_RX_DV[1,2], ENET_RX_ER[1,2], ENET_RX_CRS[1,2], ENET_RX_COL[1,2], ENET_RXD[1,2],	1)ETHERNET_IF, 2)VIRTEX1000	Signale zwischen Ethernet MAC und Ethernet PHY für Empfangen.
ENET_RESET_N[1,2], ENET_PWRDWN[1,2], ENET_PAUSE[1,2], ENET_SLEEP[1,2], ENET_LED/CFG[1,2],	1)ETHERNET_IF, 2)VIRTEX1000	Steuersignale für Intel LXT971A.
PCI_CLK[1,2], PCI_RST_N[1,2], PCI_AD[1,2], PCI_CBE[1,2], PCI_FRAME_N[1,2], PCI_IRDY_N[1,2], PCI_DEVSEL_N[1,2], PCI_TRDY_N[1,2], PCI_STOP_N[1,2], PCI_PAR[1,2], PCI_PERR_N[1,2], PCI_SERR_N[1,2], PCI_LOCK_N[1,2], PCI_IDSEL[1,2], PCI_REQ_N[1,2], PCI_GNT_N[1,2], PCI_INTA_N[1,2], PCI_INTB_N[1,2], PCI_INTC_N[1,2], PCI_INTD_N[1,2], PCI_REQ64_N[1,2], PCI_M66EN[1,2],	1)PMC_IF, 2)VIRTEX1000	PCI Signale. PCI_REQ64_N und PCI_M66EN sind pro forma an den FPGA geführt. 64-Bit-Transfer ist mit diesem Design nicht möglich. 66MHz Betrieb ist nicht vorgesehen aber für die Zukunft denkbar.
PMC_I/O[1,2]	1)PMC_IF, 2)VIRTEX1000	Verbindet die User I/Os der PMC-Schnittstelle mit dem FPGA.

Tabelle 3-1 Signalbeschreibung des Top-Levels

Seite 2

Digitale Bauteile benötigen eine Versorgungsspannung, die nur in kleinen Grenzen schwanken darf, um eine korrekte Funktion zu gewährleisten. Gerade bei Schaltvorgängen findet in kurzer Zeit eine Veränderung im Strombedarf statt. Da aber die Spannungsversorgung nicht ohne Zeitverzögerung auf diesen erhöhten Bedarf reagieren kann, d.h. genug Strom zu Verfügung stellt, führt dies eventuell dazu, dass die Spannung kurzzeitig zusammenbricht, und die Funktionsfähigkeit des Bauteils nicht gewährleistet ist. Um dies zu verhindern werden sog. Bypass-Kondensatoren verwendet, die zwischen Versorgungsspannung und Masse geschaltet werden. Diese haben keinen Einfluss auf die Funktionalität der Komponente und dienen als lokaler Energieträger. Sie können keine

Gleichspannung liefern, da die gespeicherte Energie gering ist, aber kurzzeitig Energie in Form von Ladungsträgern zu Verfügung stellen.

Kondensatoren sind keine idealen Kapazitäten. Sie haben parasitäre Widerstände und parasitäre Induktivitäten. Besonders die Induktivitäten sind durch geometrische Ausdehnung von leitendem Material bestimmt. Kleine Kondensatoren und kürzere Leiterbahnen zwischen den Anschlüssen des Kondensators und dem IC verringern die Induktivität. Je kleiner die Induktivität durch Kondensator und Leiterbahn, desto schneller kann Energie vom Kondensator zum IC transportiert werden. Kurze Leitungen sind also für hohe Frequenzen wichtig, während Kondensatoren, die zum Ausgleichen niedriger Frequenzen verwendet werden, in größerem Abstand platziert werden können.

Zusammen mit dem parasitären Widerstand lässt sich ein Kondensator durch eine RLC-Serienschaltung darstellen. Diese hat eine bestimmte Resonanzfrequenz. In einem schmalen Band um diese Resonanzfrequenz kann der Kondensator effektiv als Bypass-Kondensator eingesetzt werden. Wie breit dieses Band ist, hängt von der Art des Kondensators ab. So können Tantalum-Kondensatoren über einen weiten Frequenzbereich eingesetzt werden, XR7 Kondensatoren nur über einen sehr schmalen. Da der Frequenzbereich, der von einem Bauteil benötigt wird, unterschiedlich ist, werden oft Kondensatoren für verschiedene Frequenzbereiche eingesetzt.

Strombedarf, Frequenzen, Induktivitäten und damit Kondensatoren lassen sich ausrechnen und simulieren. Dies ist aber sehr aufwendig. Einige Hersteller machen Angaben zur Beschaltung mit Bypass-Kondensatoren. Ist dies nicht der Fall, wird im Allgemeinen für ICs ein 100nF XR7-Kondensator pro VCC-Pin für hohe Frequenzen und eventuell ein Tantalum-Kondensator mit größerer Kapazität für niedrige Frequenzen verwendet.

Im vorliegenden Design sind alle Bypass-Kondensatoren auf Seite 2 von PMC_TOP_LEVEL zusammengefasst. So ist überschaubar, wie viele und welche Kondensatoren pro Versorgungsspannung benutzt werden.

Für den Ethernet PHY von Intel und das CPLD von Xilinx gibt es genaue Empfehlungen der Hersteller, welche Kondensatoren benutzt werden sollen. Für den Intel LXT971A–Baustein wird ein 10 µF Kondensator empfohlen und für jeden der vier VCC-Pins einen 100 nF Kondensator. Xilinx empfiehlt für jeden VCC-Pin (also VCCINT und VCCO) einen 10 nF und 100 nF Kondensator.

In Anlehnung an die Application Note von Xilinx „Power Distribution System Design: Using Bypassing/Decoupling Capacitors“[A6] sind die Kondensatoren für den Virtex bestimmt worden. Dabei ist von einer maximalen Auslastung des FPGAs ausgegangen worden. Die Richtlinien aus [A6] besagen, dass Kondensatoren jeder Zehnerpotenz in einem bestimmten Kapazitätsbereich vertreten sein sollen. Das dort angezeigte Beispiel hat Kondensatoren in einem Bereich zwischen 470 µF und 4,7 nF in Zehnerpotenzen abgestuft. Dabei ist nicht der Multiplikator 4,7 ausschlaggebend, sondern die Bereiche der jeweiligen Zehnerpotenz. Pro benutztem VCC-Pin soll ein Kondensator verwendet werden. Die Anzahl der Kondensatoren aus den jeweiligen Bereichen wird über eine prozentuale Verteilung geregelt. Als Faustregel gilt, dass pro Zehnerpotenz nach unten die Anzahl der benutzten Kondensatoren sich verdoppelt.

Um die Anzahl der verschiedenen Kondensatoren auf dem Board zu beschränken, werden für niedrige Frequenzen 47 µF Tantalum-Kondensatoren benutzt, für mittlere und hohe

Frequenzen 100 nF bzw. 10 nF XR7 Kondensatoren. Weil kein Kondensator über 47 µF als qualifiziertes SMD-Bauteil vorliegt, wird der Prozentanteil der 47 µF Kondensatoren erhöht. Von 100 % entfallen 15% auf 47 µF, 30% auf 100 nF und 55 % auf 10 nF.

Kapazität	Prozent	VCCINT(36 Pins)	VCCO(40 Pins)
47 µF	15%	5	6
100 nF	30%	11	12
10 nF	55%	20	22
gesamt	100%	36	40

Tabelle 3-2 Bypass-Kondensatoren für das FPGA

Für die übrigen verwendeten Bauteile liegen keine Herstellerangaben über Bypass-Kondensatoren vor. Erfahrungswerte von anderen Entwicklern zeigen jedoch, dass 100 nF Kondensatoren als Bypass-Kondensatoren gut geeignet sind. Deshalb wird pro VCC-Pin ein 100 nF Kondensator bei den Bauteilen LT1130ACSW (RS232-Treiber), AM29F016D (Flash-Memory), XC18V02 bzw. XC18V04 (Konfigurations-PROMs) und uPD434008A (SRAM) verwendet. Zusätzlich zu diesen sind in Anlehnung an das Referenz-Design des AVNET-Boards zwei 10 µF Kondensatoren den Flashs zugeordnet: Einer dem Konfigurations-Flash und einer der ROM-Bank. In Analogie dazu ist jeder SRAM Bank ein 10 µF Kondensator zugeordnet.

3.1.4.2. VIRTEX1000

(s. A-2: Blatt 3/13)

Da es unpraktisch ist, ein Symbol mit 560 Pins zu benutzen, ist das FPGA für die vereinfachte Verwendung im Schematic in mehrere Symbole zerlegt worden, die zum Anschluss der allgemeinen Funktionseinheiten dienen, mit denen das FPGA vernetzt ist. Dies sind nur Symbole, die in ihrer Gesamtheit das FPGA repräsentieren. Die Symbole zeigen nicht das Pinout des FPGA. Dies wird mit der Software festgelegt, mit der auch die Symbole erzeugt werden und ist unabhängig von diesen. D.h. man kann das Pinout verändern, ohne die Symbole und damit das Schematic ändern zu müssen.
Abb. 3-2 gibt eine Übersicht über die ungefähre Zuordnung zwischen benutzten Pins und angeschlossener Komponente. Dabei ist zu beachten, dass bestimmte Signale feste Pins seitens des Herstellers zugeordnet bekommen. Dies betrifft Signale der Spannungsversorgung, Konfiguration und der Oszillatoren. Diese sind nicht eingezeichnet.
Nicht alle farblich unterlegten I/O-Pins sind benutzt worden. Es ist zu erkennen, dass Funktionen nicht auf bestimmte I/O-Bänke verteilt sind. Da alle VCCO-Leitungen mit 3,3V getrieben werden (s.u.), ist dies ohne weiteres möglich (vgl. [D12])

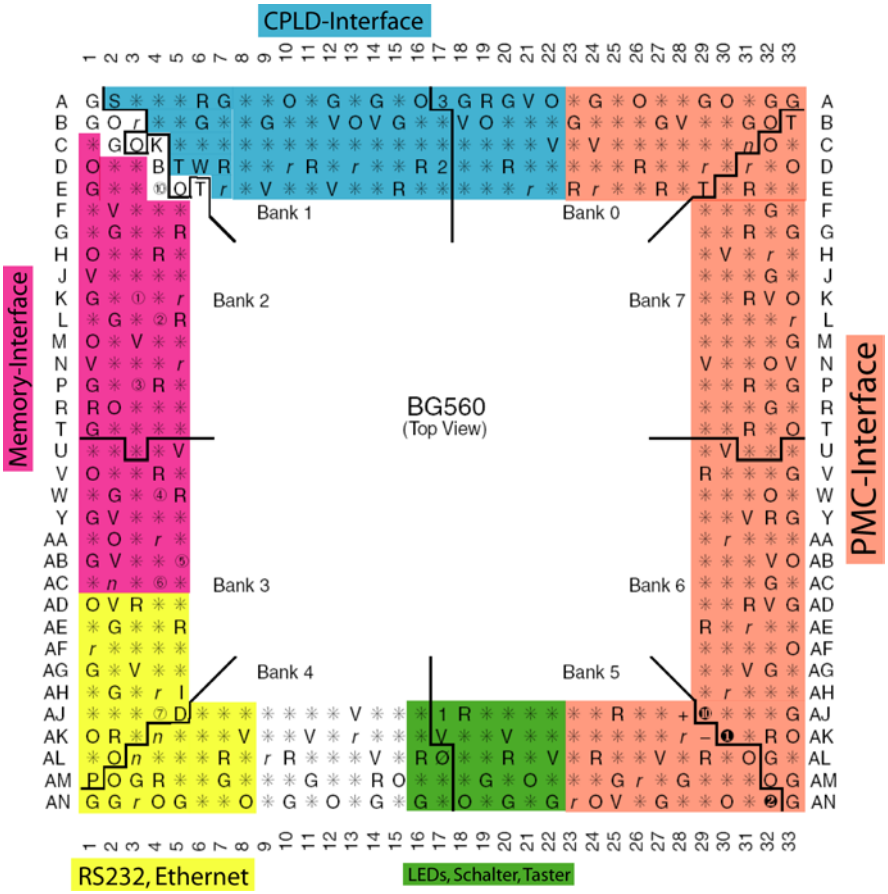


Abb. 3-2 Belegung des FPGAs nach Funktionsgruppen

Das Schematic „Virtex1000“ zeigt 8 verschiedene Symbole, die zum Anschliessen der entsprechenden Funktionseinheiten benutzt werden.

POWER/TEMP

Der Block „POWER/TEMP“ stellt die Anschlussmöglichkeiten für die interne Betriebsspannung des Virtex1000 (VCCINT), die Massepins (GND) sowie die Anschlussmöglichkeiten für die Ausgangstreiberspannung (VCCO) zu Verfügung. Außerdem stehen zwei Pins zum Anschluss eines Temperatursensors in Form einer Diode zu Verfügung.
VCCINT muss an 2,5V Spannung angeschlossen werden. VCCO ist abhängig von den I/O-Standards, die bei der Programmierung des FPGA festgelegt werden. Dies ist für das Design entweder PCI_5V oder LVTTL (Low Voltage TTL). Für beide Standards ist eine VCCO von 3,3V nötig. Es wird der PCI-5V Standard verwendet, weil dieser Ausgangspegel erzeugt, die 3,3V-PCI kompatibel sind, und im Gegensatz zum PCI_3,3V-Standard 5V tolerant ist.
Die Pins für die Temperatur-Diode werden nicht angeschlossen.

UNUSED PINS/NC

Werden im Virtex1000 I/O-Standards verwendet, die eine Referenzspannung benötigen, muss diese über spezielle Pins zu Verfügung gestellt werden. Diese Pins heißen VREF. Wird keine Referenzspannung benötigt, wie dies bei diesem Design der Fall ist, können die VREF-Pins als I/O-Ports verwendet werden. Obwohl in diesem Fall kein funktionaler Unterschied zwischen VREF- und I/O-Pins besteht, sind die nichtverwendeten Pins im Design danach getrennt im Block „UNUSED PINS/NC“ aufgeführt. Um ein Floaten zu vermeiden sind diese Pins definiert auf Masse gelegt. Zusätzlich sind die vier Pins in diesem Block enthalten, die intern keine Verbindung haben: NC (not connected).

JTAG_IF_CFG_PROM

Der Block „JTAG_IF_CFG_PROM“ stellt die Anschlüsse für die Konfigurationssignale dar. Er beinhaltet neben dem JTAG Interface auch die Ports für SelectMap und Konfigurations-Modus.Unabhängig von der Konfiguration stellt dieser Block die Ports für die Leitungen zwischen CPLD und FPGA zu Verfügung, die noch nicht funktional belegt sind.
Der Systemtakt für den LEON ist willkürlich diesem Block zugeschlagen worden.

MEM_IF

Da SRAM und ROM die gleichen Adress- und Datenleitungen benutzen, sind die Pins für diese Funktionsblöcke zusammengelegt worden zum Block „MEM_IF“.

UART_IF, LED_SW_BUT, ETHERNET_IF, PMC_IF

Stellen die Anschlüsse für die Signale der gleichnamigen Funktionseinheiten dar.

3.1.4.3. CONNECTOR_BLOCK

(s. A-2: Blatt 4/13)
Dieser Block enthält nur den Flachbandkabel-Stecker mit 34 Pins.
Zwei RS232-Schnittstellen des LEONs sind hiermit verbunden, der Treiberbaustein ist allerdings unter UART_IF zu finden. Eine verbundene UART bietet die Möglichkeit für Handshaking zwischen Sender und Empfänger durch die Signalleitungen RTS und CTS. Die andere verfügt nur über die Leitungen TX und RX. Diese wird vornehmlich durch die DSU benutzt, da sie kein Handshaking unterstützt.
22 Pins sind für die JTAG-Schnittstelle und andere Xilinx-spezifische Konfigurationssignale vorgesehen.
Die restlichen 6 Pins sind mit Masse und den unterschiedlichen Spannungen verbunden. Sie können zu messtechnischen Zwecken benutzt werden, oder um externe Spannungsversorgung anzuschließen.
Tabelle 3-3 zeigt die Zuordnung zwischen Signalen und Pins

Pin Nr.	Signalname	Beschreibung / Verwendung
1	CON_UART2_TX	RS232 Anschluss mit Handshake
2	CON_UART2_RX	
3	CON_UART2_RTS	
4	CON_UART2_CTS	
5	CON_UART3_TX	RS232 Anschluss ohne Handshake
6	CON_UART3_RX	
7	CON_JTAG_DATA(0)	JTAG-Interface und Konfigurations-signale von Xilinx Bausteinen
8	CON_JTAG_DATA(1)	
9	CON_JTAG_DATA(2)	
10	CON_JTAG_DATA(3)	
11	CON_JTAG_DATA(4)	
12	CON_JTAG_DATA(5)	
13	CON_JTAG_DATA(6)	
14	CON_JTAG_DATA(7)	
15	CON_JTAG_TDI	
16	CON_JTAG_DONE	

17	CON_JTAG_TCK	
18	CON_JTAG_PROG_N	
19	CON_JTAG_TMS	
20	CON_JTAG_INIT_N	
21	CON_JTAG_TDO	
22	CON_JTAG_BUSY/DOUT	
23	CON_JTAG_CS_N	
24	CON_JTAG_WRITE_N	
25	GND	
26	VCC	
27	GND	
28	CON_JTAG_CCLK	
29	GND	Masse
30	GND	
31	VCC	5V Versorgungsspannung
32	VCC	
33	VCC_3_3	3,3V Versorgungsspannung
34	VCC_2_5	2,5V Versorgungsspannung

Tabelle 3-3 Belegung des Flachbandkabelsteckers

3.1.4.4. POW_RES_CLK

(s. A-2: Blatt 5/13)
Der Block POW_RES_CLK enthält seinem Namen entsprechend drei Funktionseinheiten. Die benötigten Versorgungsspannungen werden generiert, Takte erzeugt und ein Reset bei Spannungsabfall aktiviert.

Spannungsgenerierung: Die 5V Versorgungsspannung wird entweder über den PCI-Bus, das Flachbandkabel oder den Jumper J1 geliefert, der auch zu Messzwecken benutzt werden kann. Eine rote SMD-LED (D1) mit Vorwiderstand (R2) zeigt das Vorhandensein einer Spannung auf der 5V Leitung an.
Von diesen 5V werden die beiden anderen benötigten Versorgungsspannungen 2,5V und 3,3V durch Linearregler erzeugt (U1, U2), die parallel geschaltet sind. Beide sind, wie vom Hersteller vorgegeben, mit Kondensatoren beschaltet (C1, C2, C3, C4). Die Ausgangsspannung (VOUT) wird durch einen Widerstandsteiler an ADJ eingestellt (s.u.). Zur Anzeige des Vorhandenseins der Spannungen wird jeweils eine rote SMD-LED (D2, D3) mit entsprechendem Vorwiderstand angeschlossen. Für Messzwecke sind beide Ausgangsspannungen und Masse an einen 6-poligen Pfostenstecker P1 angeschlossen.

Berechnung der Widerstandsteiler:

Für die Widerstandsteiler werden Widerstände mit maximal 1% Abweichung verwendet, um die Ausgangsspannung exakt einzustellen. Nach Vorgabe des Einkaufs sollen die Werte der E24 Reihe entnommen sein. **Im Folgenden wird der an VOUT liegende Widerstand mit R1, der an Masse liegende Widerstand als R2 bezeichnet**. Es fließt ein Strom aus ADJ heraus, der als Iadj bezeichnet wird und 50 uA beträgt (Herstellerangaben).
Der Hersteller gibt eine Formel zur Berechnung von VOUT an:

(A) VOUT=1,25*(1+R2/R1)+Iadj*R2

Zur einfachen Berechnung des Verhältnissen R2/R1 kann der Spannungsabfall an R2 laut Herstellerangaben vernachlässigt werden.

(B) VOUT=1,25*(1+R2/R1)
(C) R2/R1=(VOUT/1,25)-1

Der Widerstand R1 soll im 100 Ohm Bereich liegen. Dies hat zur Folge, dass Iadj * R2 klein genug wird, um vernachlässigbar zu sein.

3,3V: Aus (C): R2 = 1,64 R1
In Tabelle 3-4 sind Werte aus der E24 Reihe eingesetzt und anhand von (A) die Ausgangsspannung berechnet.

R1 / Ohm	R2 / Ohm	VOUT / V	Fehler
100	160	3,258	1,3%
110	**180**	**3,304**	0,1%
120	200	3,343	1,3%
130	220	3,376	2,3%
150	240	3,262	1,2%

Tabelle 3-4 mögliche Widerstandsteiler zur Erzeugung von 3,3V

Es wird die Kombination 110 Ohm für R1 (Bauteil R3) und 180 Ohm für R2 (Bauteil R4) gewählt.

2,5V: Aus (C) R2=R1
Da mit zunehmendem R2 der Spannungsabfall Iadj*R2 und damit der Fehler größer wird, sind kleine Widerstände vorzuziehen. Um die verwendeten Typen gering zu halten, werden für R1 und R2 (Bauteile R6, R7) jeweils 110 Ohm gewählt.

Takterzeugung: Zur Takterzeugung werden zwei SMD-Oszillatoren verwendet (U3, U4). Diese sind mit Kondensatoren nach Herstellerangaben beschaltet und haben zusätzlich Serienwiderstände am Ausgang, um eventuelle Reflexionen zu dämpfen. Der Takt, der an das FPGA geführt wird, lässt sich durch den Jumper (J2) festlegen.

Resetgenerierung: Die Spannungsüberwachung (U5) zieht den Reset Ausgang auf Low, wenn entweder die Spannung an VCC unter 3.08 V sinkt oder MR auf Low gezogen wird. Dies kann durch das CPLD gemacht werden. WDI und WDO sind Watchdog Ports, die für zukünftige Designs angeschlossen sind. PFI wird nach Herstellerempfehlung auf VCC gelegt, da die damit verbundene Funktionalität nicht benutzt wird.

3.1.4.5. JTAG_IF_CFG_PROM

(s. A-2: Blatt 6/13)
Der Block JTAG_IF_CFG_PROM hat die Aufgabe, die Konfiguration des FPGA bei Power-Up oder nach einem Reset zu gewährleisteten. Wie in Kapitel 3.1.1 dargelegt, müssen zwei verschiedene Arten der Konfiguration möglich sein: MasterSerial und SelectMap. Außerdem soll Scrubbing möglich sein. Um dies zu erreichen gibt es drei Einheiten in dieser Hierarchie. Das CPLD (U7), das Flash (U8) und die beiden Xilinx PROMs (U9, U10). Das CPLD ist durch zwei Symbole dargestellt, um die 144 Pins zu verteilen, ähnlich dem Virtex 1000 (s. Kapitel 3.1.4.2). Die beiden Xilinx PROMs sind gemäß den Herstellerangaben

zusammengeschaltet, um genügend Speicher für die Konfiguration eines Virtex1000 bereitzustellen.

Im Folgenden werden die verschiedenen Konfigurations- bzw. Scrubbingmöglichkeiten beschrieben. Tabelle 3-5 fast die Einstellungen durch Jumper zusammen.

(A)SelectMap mit Flash: Bei Konfigurierung im SelectMap Modus mit Daten des Flashs, generiert das CPLD die Adressen für den Speicher. Im SelectMap Modus wird die CCLK nicht durch das FPGA erzeugt, sondern durch einen anderen Baustein. Im vorliegenden Design macht dies das CPLD.

(B)SelectMap mit PROMs: Sollen zur Konfiguration die Xilinx PROMs benutzt werden, ist kein CPLD zur Adressgenerierung notwendig. Die PROMs unterstützen diesen Betriebsmodus. Trotzdem muss das CPLD die Signale CS_N und WRITE_N auf Low ziehen und CCLK treiben, da dies für SelectMap notwendig ist.

(C)Serielle Konfiguration mit PROMs: Seriell kann nur mit den PROMs konfiguriert werden. Das CPLD ist nicht nötig. Ob FPGA (Master Serial Mode) oder CPLD (Slave Serial Mode) CCLK treiben, ist von der Wahl des Modus abhängig.

(D)Scrubbing mit Flashdaten: Das CPLD steuert das Flash.

(E)Scrubbing mit PROMs: Das CPLD steuert die PROMs.

Die Art der Konfiguration – also (A)/(B) oder (C) - wird durch die drei Mode Jumper (J4, J5, J6) eingestellt. Die Codierung ist dem Datenblatt des Virtex zu entnehmen. Durch die Pullups (R15, R16, R17) wird die Default-Einstellung auf Slave Serial Mode gesetzt.

Um zwischen (A) und (B) zu wählen, ist - neben der richtigen Einstellung der Mode Jumper - der Jumper J3 zuständig. Welche Einstellung welchen Speicher auswählt, ist nicht durch das Design festgelegt. Dies muss die Konfiguration des CPLD festlegen. Dieser Jumper wird auch benötigt, um zwischen (D) und (E) zu wählen.

Um den Port CCLK der CPLD physikalisch vom CCLK Pin des FPGA trennen zu können, ist der Jumper J9 eingesetzt worden. Wird dieser Jumper offengelassen, kann nicht der Zustand eintreten, dass sowohl das CPLD als auch FPGA das Signal CCLK treiben. Dies könnte bei falsch konfiguriertem CPLD im Falle (C) mit Einstellung der Mode Jumper auf Master Serial Mode passieren.

Die Wahl, ob das CPLD oder das FPGA die PROMs steuert, wird über die Jumper J7 und J8 getroffen. Diese stellen ein, welcher Baustein elektrisch verbunden wird. Im Falle des FPGA wird nach Herstellerangaben OE/RESET der PROMs mit INIT_N verbunden und CE mit DONE, im Falle des CPLD werden die entsprechenden Steuerports verbunden. Z.B. ist für (E) eine Steuerung der PROMs durch das FPGA zwingend erforderlich.

Um manuell eine Neukonfiguration zu starten, ist der Taster B1 vorgesehen. Dieser ist an das Signal PROG_N angeschlossen, und zieht dieses bei Betätigung auf Low. Daraufhin wird das FPGA neu konfiguriert.

Während eines Programmiervorgangs zieht das FPGA das Signal DONE auf Low. Nach Beendigung der Konfiguration wird DONE wieder auf High gesetzt, um das Ende

anzuzeigen. Um dies auch optisch dem Benutzer sichtbar zu machen, wird die rote LED D4 mit Vorwiderstand verwendet.

Art der Konf.	Jumper-Settings (X=gesetzt, 0=nicht gesetzt D=don't care, ?=abh. von CPLD-Programmierung, 1=Verbindung PROM-CPLD, 2=Verbindung PROM-FPGA)						mögliche Scrubbing-Art	Anmerkungen
	J4	J5	J6	J3	J9	J7, J8		
(A)	0	0	X	?	X	1	(D)	SelectMap mit Flash. PROMs müssen von CPLD deaktiviert werden.
	X	0	X					
(B)	0	0	X	?	X	1	(E)	SelectMap mit PROMs. CPLD steuert PROMs.
	X	0	X					
(B)	0	0	X	?	X	2	n/a	SelectMap mit PROMs. FPGA steuert PROMs.
	X	0	X					
(C)	X	X	X	D	0	1	n/a	Master Serial. CPLD steuert PROMs.
	0	X	X					
(C)	X	X	X	D	0	2	n/a	Master Serial. FPGA steuert PROMs.
	0	X	X					
(C)	0	0	0	D	X	1	n/a	Slave Serial. CPLD steuert PROMs.
	X	0	0					
(C)	0	0	0	D	X	2	n/a	Slave Serial. FPGA steuert PROMs.
	X	0	0					

Tabelle 3-5 Jumpereinstellungen zur Konfiguration und Scrubbing

Das CPLD muss aus den Einstellungen der Jumper J4, J5, J6, J3 erkennen, welche Art der Konfiguration gewählt ist. Allerdings kann darüber nicht festgestellt werden, ob bei serieller Konfiguration die CE- und OE-Pins der PROMs an das CPLD oder das FPGA angeschlossen sind. Sollte also die Steuerung der PROMs durch das CPLD benutzt werden, muss die Funktionalität des CPLD dies gewährleisten.

Die Konfigurationssignale PROG_N, DONE, INIT_N, BUSY/DOUT, CCLK, CS_N und WRITE_N müssen vom CPLD benutzt werden, um eine korrekte Ansteuerung des FPGAs, des Flashs oder der PROMs zu erzeugen. Bei Bedarf wird CCLK mit dem anliegenden 40 MHz Takt getrieben.

Neben den Signalen zur Konfiguration und Ansteuerung kann das CPLD einen Reset auslösen (über MR), den Watchdog der Spannungsüberwachung (U5 in POW_RES_CLK) benutzen und mit dem FPGA über 60, bisher nicht verplante Leitungen kommunizieren. Hierüber hat das FPGA auch Zugriff auf alle vom CPLD aus gesteuerten Signale.

Alle Funktionen müssen durch noch nicht vorhandenen HDL-Code für das CPLD beschrieben werden. Dies ist nicht im Rahmen der vorliegenden Arbeit enthalten.

Konfigurieren des CPLD: Das CPLD wird über das serielle JTAG Interface konfiguriert. Es ist das erste Glied in der JTAG-Kette.

Beschreiben des Konfigurations-Flash: Da das Flash nur über das CPLD adressierbar und steuerbar ist, muss ein spezielles CPLD-Design benutzt (bzw. entwickelt) werden, das ein Programmieren des Flashs mit Daten erlaubt, die z.B. über die JTAG-Schnittstelle seriell übertragen werden. Dies wird unter anderem im Referenz Design des AVNET-Boards so gemacht.

Beschreiben der Xilinx PROMs: Die PROMs von Xilinx sind speziell für Konfiguration von FPGAs entwickelt worden. Deshalb haben sie ein JTAG-Interface, über das sie beschrieben werden können. Sie sind hinter dem CPLD und vor dem FPGA in die JTAG-Kette eingeordnet.

3.1.4.6. SRAM

(s. A-2: Blatt 7/13 und 8/13)
Das Schematic des SRAM besteht aus zwei Seiten, die jeweils eine SRAM-Bank enthalten. Der Aufbau der Bänke ist identisch. Pro Bank sind jeweils vier SRAM-Bausteine zu einer Wortbreite von 32-Bit zusammengeschaltet. Jedes Bauteil ist an 8 Bit angeschlossen. Die Adressen sind für alle Komponenten die gleichen. Zwischen den Bänken wird durch das Signal RAMS_N gewählt. Beim Lesen wird das entsprechende Bit des Signals RAMOE_N aktiviert. Um byteweises Schreiben zu unterstützen, bekommen die 4 Komponenten pro Bank je ein eigenes Write Enable (WE) Signal zugeordnet, die zum Port-Signal RAMWE_N zusammengefasst sind.
Aufgrund der Größe der Speicherbausteine werden nur 19 der 21 Adresssignale benutzt. Insgesamt stehen 4 MB Speicherkapazität zu Verfügung.

3.1.4.7. ROM

(s. A-2: Blatt 9/13)
Das ROM besteht aus einer Bank. Vier Flash-Memories werden zu einer Wortbreite von 32 Bit zusammengeschaltet. Jede Komponente ist an 8 Bit angeschlossen. Alle Bausteine teilen denselben Adressbus und dieselben Steuersignale ROMS_N, ROMOE_N, WE_N, RESET_N. Die Ausgänge RY/BY jeder Komponente sind gemäß den Herstelleranweisungen zusammengeschaltet und mit einem Pullup Widerstand versehen. Die Verwendung dieses Signals ist optional. Das RESET_N Signal ist an den Reset der Spannungsüberwachung (U5 in POW_RES_CLK) angeschlossen. Bei Aktivierung des Resets oder Power-Up wird das Flash in den Lese Modus gesetzt, so dass Daten ausgelesen werden können. Das Beschreiben muss durch das FPGA geschehen, da dieses die Steuersignale treibt. Der LEON bietet Möglichkeiten, während des Betriebes sein ROM umzuprogrammieren. Soll das Flash ohne Verwendung des LEON beschrieben werden, muss dafür ein HDL-Code benutzt oder entwickelt werden.

3.1.4.8. UART_IF

(s. A-2: Blatt 10/13)
Der Block UART_IF enthält den Treiberbaustein für die RS232-Leitungen (U22), die DSUB9 Schnittstelle (P3) und Kondensatoren (C8, C9, C10, C11, C12, C13).

Der Treiberbaustein LT1130A bietet 5 Sendeleitungen und fünf Empfangsleitungen. Dies reicht genau für die drei UARTs des LEON aus. Das Bauteil ist nach Herstellerangaben mit Kondensatoren beschaltet, die zur internen Spannungserzeugung benutzt werden.

Die UART-Signale, die bei der späteren Implementierung auf die UART1 Ports gemappt werden, gelangen an die DSUB9 Buchse. Die anderen beiden UARTs werden aus diesem Block heraus an den Block CONNECTOR_BLOCK übergeben und dort an den Flachbandkabel-Stecker.

3.1.4.9. LED_SW_BUT

(s. A-2: Blatt 11/13)
Der Block LED_SW_BUT enthält 8 grüne Leuchtdioden mit Vorwiderständen, 8 Schalter mit Pullups und vier Taster mit Pullups und Entprell-Kondensatoren, die sich an dem Referenz Design von AVNET orientieren. Alle Bauteile sind zu Testzwecken vorhanden. Welche Funktionen sie erfüllen, ist durch FPGA-Konfiguration bestimmt.

3.1.4.10. ETHERNET_IF

(s. A-2: Blatt 12/13)
Der Block ETHERNET_IF beinhaltet die Ethernet PHY von Intel (U24), die RJ45-Buchse (P4), drei LEDs (D113, D14, D15) und durch Hersteller und Referenzdesign vorgegebene Widerstände, Kondensatoren, Jumper und Ferrit Perlen.
Das Design dieser Hierarchieebene orientiert sich an dem Referenz-Design des AVNET Boards. Dieses wurde bezüglich seiner Ethernet-Schnittstelle schon getestet und ist deshalb einsetzbar. Die Änderungen im Gegensatz zum AVNET-Board sind im Folgenden aufgelistet.

JTAG-Interface: Nach Vorgabe von Astrium soll der Ethernet PHY nicht über JTAG programmiert werden können. Deshalb wird die Schnittstelle offengelassen. Eine definierte Einstellung von Pegeln an den Pins ist nach Herstellerangaben nicht nötig. Floaten deaktiviert die Schnittstelle.

1%-Widerstände: Im Referenzdesign sind drei Widerstände mit 1% Abweichung verwendet. Nach Vorgabe von Astrium werden nur 1% - Widerstände aus der E24 Reihe benutzt.

AVNET	Intel	Astrium
22,1 kOhm	22,1 kOhm	22KOhm (R55)
49,9 Ohm	50 Ohm	2*100 Ohm parallel (R56, R57)
49,9 Ohm	50 Ohm	2*100 Ohm parallel (R58, R59)

Tabelle 3-6 Vergleich der Ethernet PHY Beschaltung

Status-LEDs: Im Referenzdesign wird eine RJ45-Buchse mit zwei integrierten LEDs benutzt. Im vorliegenden Design wird stattdessen eine RJ45-Buchse ohne LEDs benutzt, da diese Buchsen in flacheren Versionen bestellbar sind. Zusätzlich werden drei 5mm LEDs in den Farben rot, grün, gelb mit Vorwiderständen verwendet. Diese sind am Front Bezel platziert, um von außen sichtbar zu sein. Gesteuert werden sie im Betrieb von dem Ethernet PHY.

3.1.4.11. PMC_IF

(s. A-2: Blatt 13/13)
Der Block PMC_IF enthält die PMC-Schnittstelle. Da kein 64 Bit PCI-Betrieb vorgesehen ist, fällt der Stecker Pn3 der CMC-Spezifikation (s. Kapitel 2.1.3.2) weg. Die Stecker Pn1 und Pn2 werden für den PCI-Bus benötigt. Der Stecker Pn4 stellt vom Anwender zu definierende I/O-Leitungen zu Verfügung, die über den Bus PMC_I/O an das FPGA geführt sind.

Fast alle PCI-Signale sind direkt zum FPGA geführt. Ausnahme bilden die optionalen PCI-Signale M66EN und PME#. PME# (Power Managment Event) wird weder von der AMBA-PCI-Bridge, vom LEON noch von den Referenz Designs benutzt. Dieser Pin wird offen gelassen. M66EN ist, durch einen Jumper (J19) unterbrochen, an das FPGA gelegt. So kann in eventuellen zukünftigen Designs, die 66 MHz PCI unterstützen, diese Leitung verbunden werden.

Die BUSMODE# Signale der CMC-Spezifikation sind zur Identifizierung von Mezzanine Karten vorgesehen. Da das BUSMODE#(3) Signal auf das BUSMODE#(1) Signal gelegt ist, wird die PMC-Karte einem Host (z.B. dem Trägerboard) signalisieren, dass eine Karte den PMC-Steckplatz belegt und das PCI Protokoll unterstützt und benutzt.

Die PMC-Schnittstelle stellt ein JTAG-Interface zu Verfügung. Dieses wird hier nicht benutzt. Um eine eventuell vorhandene JTAG-Kette nicht zu unterbrechen, ist das Signal TDI mit TDO verbunden.

Sowohl die 5V als auch die 3,3V Spannungsleitungen des PCI-Bus werden analog zum Referenz Design FME mit Kondensatoren gepuffert. Die 5V Leitung geht immer an die 5V Ebene der PMC-Karte. Die 3,3V Leitung ist optional über den Jumper J20 an die 3,3V Ebene der PMC-Karte anschließbar.

3.2. PCI-Interrupt Controller

Im Rahmen der Diplomarbeit soll ein einfacher Interrupt-Controller entwickelt werden, der die Weiterleitung von Interrupts auf dem PCI-Bus an den LEON gewährleistet. Diese Funktionalität wird durch den verwendeten IP-Core der AMBA-PCI-Bridge nicht zu Verfügung gestellt.

3.2.1. PCI-Interrupt Leitungen

Auf dem PCI-Bus stehen immer vier Interrupt Leitungen zu Verfügung, die allen angeschlossenen Komponenten zugänglich sind. Im Folgenden werden diese Leitungen INTW, INTX, INTY, INTZ genannt. Beim PCI-Steckplatz heißen die Pins, die an die Leitungen angeschlossen sind, INTA, INTB, INTC, INTD. Wie diese Leitungen auf dem Trägerboard jedoch verlaufen, ist nicht durch die PCI-Spezifikation vorgegeben. D.h., dass nicht alle Komponenten ihr INTA an INTW angeschlossen haben, sondern z.B. Komponente 1 INTA, Komponente 2 INTB, Komponente 3 INTC usw.. Abb. 3-3 zeigt eine mögliche Form der Vernetzung.

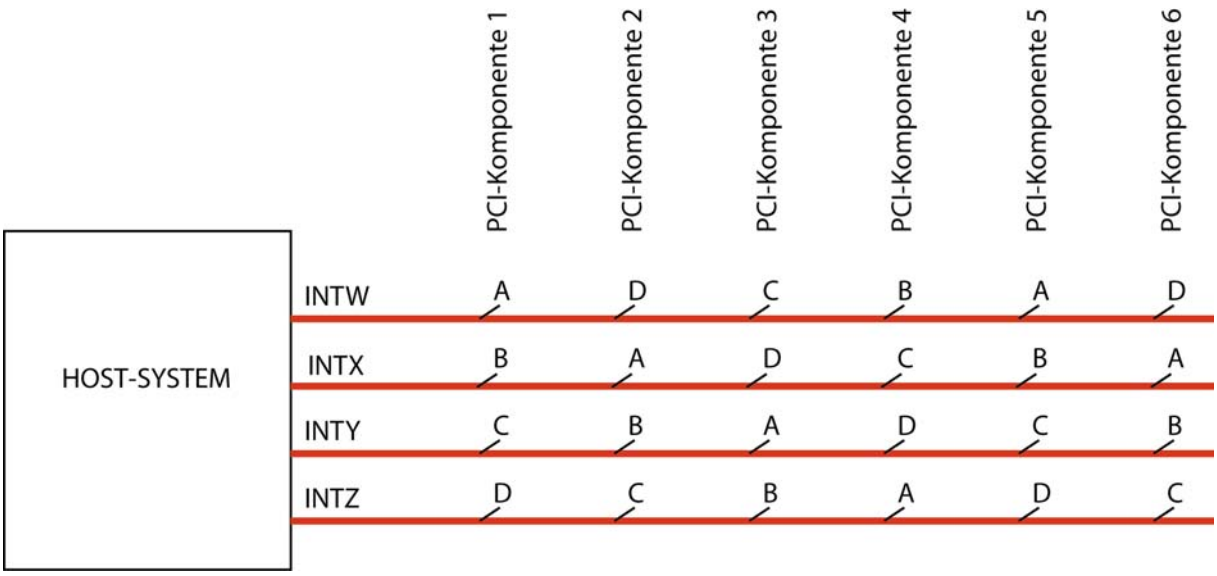


Abb. 3-3 Beispielhaftes PCI-Interrupt Routing

PCI-Komponenten mit nur einer Funktion können nur einen Interrupt benutzten. Dieser muss immer INTA sein. PCI-Komponenten mit mehreren Funktionen können mehrere Interrupts benutzten. Es wird deutlich, dass das Hostsystem nicht zwangsläufig anhand INTW, INTX usw. entscheiden kann, welche Komponente einen Interrupt signalisiert. Dies muss durch Softwareabfragen entschieden werden. PCI-Interrupts sind low-aktiv.

3.2.2. PCI-Interrupt Controller

Der LEON muss nach Vorgabe von Astrium als Host eines Systems aber auch als Prozessor ohne Hosteigenschaften einsetzbar sein. Dies führt zu unterschiedlichen Anforderungen an den Interrupt Controller. Dies ist im Folgenden erläutert. Zur Unterscheidung zwischen diesen beiden Betriebsarten dient die schon vorhandene Variable HOST_BRIDGE_c der AMBA-PCI-Bridge.

3.2.2.1. LEON als Host

Ist der LEON das Hostsystem, können auf allen vier Leitungen INTW, INTX usw. Interruptsignale von PCI-Komponenten gesendet werden. Diese werden über die Pins INTA, INTB usw. an das FPGA weitergegeben. Wird der LEON mit nur einem internen Interrupt Controller benutzt, stehen die Interrupts IRQ 12–15 noch zu Verfügung, die high-aktiv sind. Dies würde die vier PCI-Interrupts abdecken. Das LEON Manual warnt jedoch davor, den IRQ 15 zu benutzen, da dieser nicht *maskable* ist. Deshalb wird von der Benutzung dieses Interrupts abgesehen.
Stattdessen wird nur ein Interrupt des LEONs benutzt, um alle vier PCI-Interrupt-Leitungen anzuzeigen. D.h. INTA, INTB usw. werden NAND-verknüpft und an IRQ14 gelegt (IRQ14 ist vom LEON-Core dafür vorgesehen). Siehe auch Abb. 3-5.
Ein Register speichert synchron die negierten Zustände der PCI-Interrupt-Leitungen, die vom LEON abgefragt werden können (s. Abb. 3-4).

3.2.2.2. LEON ohne Hosteigenschaften

Ist der LEON nicht das Hostsystem, muss er die Möglichkeit haben, dem Hostsystem eine Interrupt Anforderung zu signalisieren. Dafür bekommt er einen Interrupt auf dem PCI-Bus zugewiesen. Laut Spezifikation muss dies INTA sein.
Der LEON-VHDL-Core hat kein Signal dafür vorgesehen, einen Interrupt auf den PCI-Bus zu signalisieren. Dies muss durch zusätzliche Funktionalität ermöglicht werden. Es wird ein Bit in einem Register vorgesehen, das vom LEON gesetzt und gelöscht werden kann. Ist es gesetzt, wird der Interrupt auf INTA aktiviert, wird es gelöscht, wird er deaktiviert (s. Abb. 3-4).

3.2.2.3. Einbindung in die AMBA-PCI-Bridge

Der PCI-Interrupt Controller wird in die AMBA-PCI-Bridge als Komponente eingebunden. Dies hat den Vorteil, dass ein Register leicht eingefügt werden kann, weil eine Schnittstelle zum Beschreiben und Lesen schon existiert.

Das Kontroll Register für den Interrupt Controller (irq_reg) wird in den Konfigurationsbereich des PCI-Interfaces der Bridge an Adresse 0x40 implementiert. Dies ist ein anwenderspezifischer Adressbereich, der sowohl von PCI-Seite als auch von APB-Seite adressierbar ist. Bei einem Reset werden alle Bits des Registers gelöscht. Abb. 3-4 zeigt das Kontroll Register.

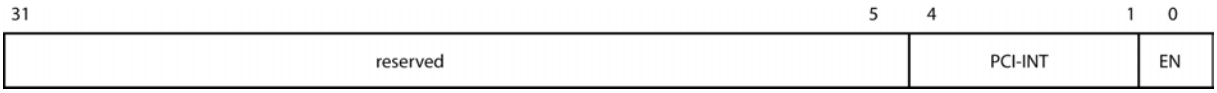


Abb. 3-4 Interrupt Kontroll Register der AMBA-PCI-Bridge (irq_reg)

- PCI-INT (read only): Speichert synchron die PCI-Interrupt Leitungen. Wenn gesetzt, ist der zugehörige Interrupt aktiv. INTA→Bit0, INTB→Bit1 ...
- EN: Enable. Wenn gesetzt, wird bei Betrieb des LEON ohne Hosteigenschaften INTA aktiviert.

INTB, INTC und INTD sind immer Eingangssignale. INTA ist in Abhängigkeit von HOST_BRIDGE_c entweder Ein- oder Ausgangssignal. Es wird im LEON ein *geniodpad* als Pad instantiiert. Dies ist ein inout opendrain Pad. Liegt eine Eins am Port d an, wird der Ausgang hochohmig geschaltet. Dies ist für den Betrieb als Host nötig. Liegt an d eine Null an, wird der Ausgang auf Low gezogen.
Um zu verhindern, dass PCI-Interrupts bei Betrieb ohne Hosteigenschaften an den LEON weitergegeben werden, ist in Abhängigkeit der Konstante HOST_BRIDGE_c ein Multiplexer eingefügt. Abb. 3-5 zeigt das Blockschaltbild des PCI-Interrupt Controllers.

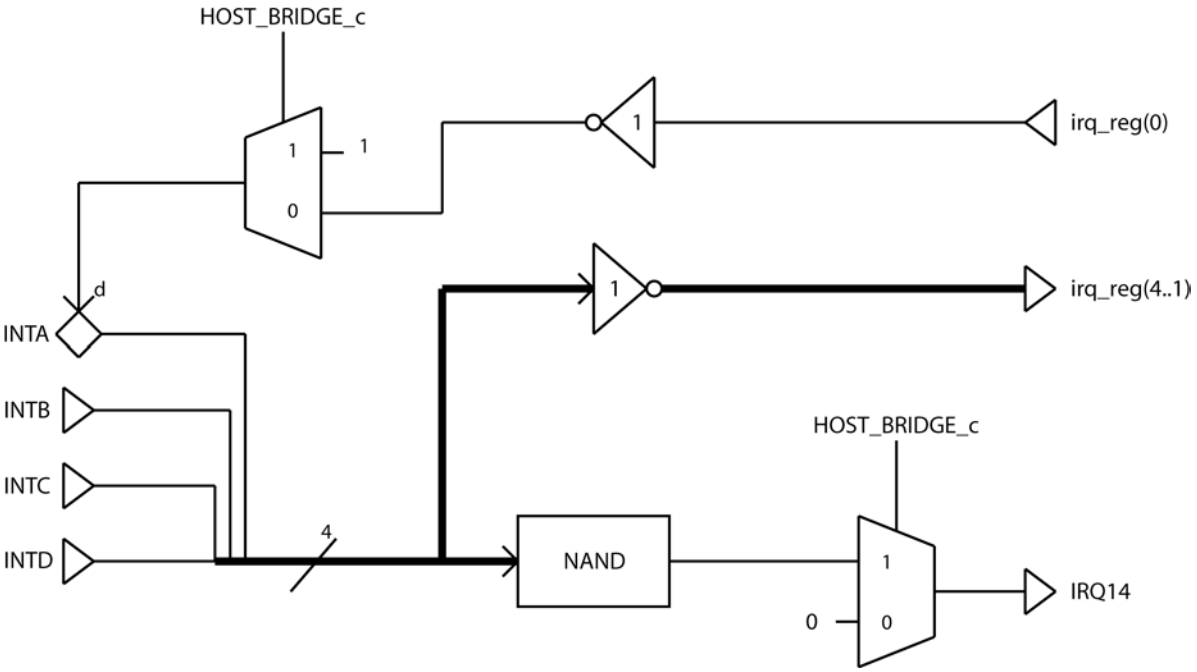


Abb. 3-5 Blockschaltbild des PCI Interrupt Controllers

4. Implementierung und Test des LEON auf AVNET-Board

Das LEON-Design wird inklusive AMBA-PCI-Bridge auf einem Evaluierungs-Board der Firma AVNET getestet (s. Kapitel 2.1.1). Das LEON-Design bietet eine große Möglichkeit an Konfigurationen. Die gewählte wird in Kapitel 4.1.1 beschrieben.
Es werden verschiedene Tests durchgeführt, die einerseits die Funktionsfähigkeit der Schnittstellen des Boards sicherstellen sollen, andererseits aber auch die Funktionsfähigkeit des implementierten Designs überprüfen. Die Ergebnisse können für spätere Tests des PMC-Boards als Vergleich herangezogen werden.
Testumgebung und Tests werden in Kapitel 4.2 beschrieben. In Kapitel 4.3 werden Tests beschrieben, die in Zukunft noch durchgeführt werden müssen, um die volle Funktionsfähigkeit des implementierten LEONs zu überprüfen.

4.1. Implementierung

Das Design wird mit dem Synthese-Tool Synplify 7.2 synthetisiert und im Anschluss mit der ISE-Entwicklungsumgebung 5.2 von Xilinx weiterverarbeitet.
Die zeitlichen Anforderungen an den Place & Route Prozess sind auf der einen Seite durch den Systemtakt des LEON, auf der anderen Seite durch die Zeitkriterien des PCI-Busses gegeben. Als Systemtakt für den LEON stehen vier verschiedene Taktfrequenzen zu Verfügung: 24,576 MHz, 27 MHz, 33 MHz und 40 MHz. Um die Anforderungen niedrig zu halten, wird der langsamste Takt verwendet (24,576 MHz). Der PCI-Bus arbeitet mit 33 MHz und die PCI-Spezifikation stellt für fast alle Bussignale folgende Anforderung: Signale müssen vom Sender spätestens 11 ns nach PCI-Taktflanke gültig sein und empfangene Signale sind frühestens 7 ns vor der Taktflanke als gültig zu betrachten.
Es werden keine räumlichen Vorgaben durch Floorplaning gemacht.
Das Pinout wird dem AVNET-Board angepasst. Der vorhandene Flash-Speicher wird als ROM Bank 1 verbunden, 6 der 8 freien PIO-Ports werden auf DIP-Schalter gelegt und zwei auf LEDs. Zwei DIP-Schalter dienen als DSUEN und DSUBRE Signal, eine LED als Anzeige für Debug-Mode. Ein Push Button wird als Prozessor Reset verwendet. UART 1 wird auf die auf dem AVNET-Board vorhandene RS232 Schnittstelle gelegt, UART 2 und DCL auf die Ports, die mit einem externen Board mit zwei RS232-Treibern und -Schnittstellen verbunden sind. Das SRAM-Interface wird nicht benutzt, da nur SDRAM auf dem Evaluations-Board vorhanden ist. Dieses wird an das SDRAM-Interface des LEON angeschlossen.

Der Place & Route-Prozess kann ohne Floorplaning und mit einem „normalen Aufwandslevel“ des Tools (Einstellung für Place&Route-Prozess der Xilinx-Entwicklungsumgebung) durchgeführt werden. Drei Kriterien müssen erfüllt sein, um den Prozess als erfolgreich abgeschlossen zu betrachten. Nach Durchlaufen des Place & Route-Prozesses müssen die in Klammern angegebenen Meldungen vorliegen:

- Alle Elemente müssen platziert sein (Placement: no errors)
- Alle Verbindungen müssen hergestellt worden sein (Routing: no errors)
- Alle Zeitkriterien müssen eingehalten werden (Timing: no errors)

Das durch die ISE-Entwicklungsumgebung erzeugte BIT-File kann auf zwei verschiedene Arten zum Programmieren des FPGAs benutzt werden. Ist zur Generierung des BIT-Files als Startup Clock der JTAG-Takt vorgegeben worden, wird mit dem BIT-File über ein serielles

JTAG-Interface das FPGA direkt programmiert. Dafür muss eine Spannungsversorgung des Testboards gewährleistet sein. Diese Art der Implementierung ist nicht resistent gegen Spannungsverlust.
Wird statt des JTAG-Taktes CCLK als Startup Clock vorgegeben, kann das erzeugte BIT-File dazu benutzt werden, ein für diese Zwecke vorgesehenes Flash auf dem Board zu programmieren. Als Ansteuerung für das Flash dient dabei ein CPLD, das vorher mit einem durch AVNET mitgelieferten Design programmiert wurde. Das CPLD verarbeitet Daten, die über die serielle JTAG-Schnittstelle geliefert werden, und wandelt sie in ein adäquates Format für das Flash um. Im Lieferumfang des Evaluations-Boards ist zusätzlich die PC-Software enthalten, mit der das BIT-File in einen seriellen Datenstrom für das CPLD umgewandelt und über das JTAG-Kabel gesendet wird. Diese Art der Programmierung dauert ca. 3 Stunden. Grund für die lange Programmierzeit ist nach Herstellerangaben die verwendete Software. Danach wird das FPGA bei Power-Up automatisch mit den Daten des Flash konfiguriert.

4.1.1. Konfiguration des LEON

Der LEON ist in weiten Teilen konfigurierbar entwickelt worden. D.h., dass durch Variablen vor der Synthese festgelegt werden kann, wie das spätere Design werden soll. Dies betrifft vor allem:

- Verwendete Komponenten: Der Anwender kann darüber entscheiden, welche peripheren Komponenten mit dem LEON implementiert werden. Zweiter IRQ-Controller, DSU, PCI-Bridge, PCI-Arbiter, AHB-RAM, AHB-Statusregister, AHB-Writeprotection und LEON Konfigurationsregister sind optional.
- Adressbereiche: Auf dem AHB- und APB-Bus können die Adressbereiche für die einzelnen Komponenten festgelegt werden.
- IU-Konfiguration: Es kann konfiguriert werden, ob und welcher Hardware-Multiplizierer und Hardware-Dividierer implementiert werden soll. Es kann festgelegt werden, wie viel Registerfenster und Hardware Watchpoints implementiert werden. Ob ein Co-Prozessor oder eine Fließkomma-Arithmetik an die IU angeschlossen ist, ist konfigurierbar.
- Cache-Konfiguration: Es ist für Befehls- und Datencache unabhängig konfigurierbar, wie viele Sets von welcher Größe implementiert werden und welche Art von *Replacement Policy* sie haben sollen.
- Synthese: Es kann festgelegt werden, ob technologieabhängig spezielle Zellen (ROM, RAM, PADS) direkt implementiert werden - d.h. das Synthese-Tool hat keine Entscheidungsfreiheit - oder ob dem Synthesetool die Instantiierung der technologieabhängigen Elemente überlassen wird.
- Boot Konfiguration: Es kann festgelegt werden, von welchem Ort aus gebootet wird. Entweder wird ein sog. internes Boot PROM instantiiert, oder aus externem Speicher gebootet. Es ist auch möglich, durch Wahl von PIO(4) zwischen extern und intern zu unterscheiden. Das sog. interne Boot PROM ist nicht im eigentlichen Sinne ein PROM. Es ist ein durch ein VHDL-File emuliertes PROM, das aus C-Code oder Assembler erzeugt werden kann (s. Kapitel 4.2).

Der im Test benutzte LEON hat folgende peripheren Komponenten:
DSU, PCI-Bridge und LEON-Konfigurationsregister. Es werden die Standard-Adressbereiche verwendet (s. Kapitel 2.2.1.3). Für die PCI-Bridge werden die Adressbereiche 0x80000100 – 0x800001FC (APB-Bus) und 0xA0000000 – 0xFFFFFFFC (AHB-Bus) verwendet.

Die IU wird mit 8 Registerfenstern und 4 Watchpoints implementiert.
Sowohl Befehls- als auch Datencache werden mit zwei Sets mit je 1 kB Speicherkapazität implementiert. Die *Replacement Policy* ist *random*.
Die Zieltechnologie ist die Virtex Reihe. Das Instantiieren von RAM und Registersätzen wird direkt gemacht, alles andere kann das Synthese-Tool instantiieren.
Es wird von einem internen PROM gebootet.

Die instantiierte PCI-Bridge hat auch Konfigurationseinstellungen. Folgende werden verwendet:

- Die Bridge ist keine Host-Bridge (s. Kapitel 4.2).
- Die Bridge identifiziert sich anderen PCI-Komponenten als Co-Prozessor.
- Die Bridge zeigt dem PCI-Bus zwei dahinterliegende Adressbereiche an: Einen 64 MB großen und einen 256 MB großen Bereich. Wird der 64 MB-Adressbereich über den PCI-Bus adressiert, wird automatisch der Adressbereich des AHB-Busses ab 0x40000000 adressiert. Dies entspricht dem SDRAM. Der 256 MB große Adressbereich wird automatisch in einen Zugriff auf den AHB-Adressbereich 0x80000000 – 0x8FFFFFFC übersetzt. Dies entspricht dem Adressraum der APB-Bridge. Dadurch kann auf die Kontrollregister der On-chip-Peripherie des LEONs zugegriffen werden.

4.1.2. Änderungen am Source-Code

Der Source-Code des LEONs stellt ein Interface für die AMBA-PCI-Bridge zu Verfügung und instantiiert die I/O-Pads selbstständig. Signale, die an die Bridge geführt werden, sind unidirektional. Die im ursprünglichen VHDL-Core der Bridge vorhandenen I/O-Zellen sind entfernt worden, so dass nur noch unidirektionale Ports im Top-Level der Bridge vorhanden sind. Die Port-List des Top-Level-Designs des LEON wird um die noch nicht vorhandenen PCI-IRQ Ports INT A – INT D erweitert. Auch die in der Hierarchie darrunterliegenden Komponenten „mcore“ und „pci“ werden um diese Ports erweitert.

4.2. Test

Es müssen verschiedene Tests durchgeführt werden, um die Funktionsfähigkeit des Systems zu verifizieren. Dabei kann zwischen verschiedenen Testszenarien – d.h. Testumgebungen – und verschiedenen zu testenden Komponenten unterschieden werden.
Als Testszenarien kommen drei in Frage:

- Das AVNET-Board wird „*stand alone*“ betrieben. D.h. Spannung wird über einen Spannungsgenerator geliefert und der PCI-Bus ist nicht angebunden.
- Der LEON arbeit als Nicht-Host-Prozessor in einem PCI-Bus-System. Dann übernimmt der Host-Prozessor die Adressverwaltung und die Bedienung von PCI-IRQs. Für dieses Szenario ist ein normaler PC geeignet, da er normalerweise freie PCI-Steckplätze zu Verfügung stellt.
- Der LEON ist der Host-Prozessor. Er konfiguriert alle PCI-Komponenten und übernimmt die PCI-IRQ-Verarbeitung. Ein PC ist ungeeignet als Testumgebung, da die CPU eines PCs immer der Host ist.

Die Funktionen, die im Stand-Alone-Betrieb des Boards getestet werden können, werden durch Testszenarien mit Anbindung an den PCI-Bus vollständig abgedeckt. Da die Implementierung der Bridge keine Probleme bereitet (s. Kapitel 4.1), ist es nicht notwendig, dieses Testszenario zu durchlaufen.
Ein Unterschied im Betrieb des LEONs als Host oder als Nicht-Host besteht vor allem seitens des Aufwandes an Software. Im Betrieb als Host muss die Verwaltung des gesamten Systems durch den LEON gewährleistet sein, was eine bestimmte Komplexität an Software voraussetzt.
Neben Software-Aspekten gibt es auch Unterschiede in den Anforderungen an die Funktionalität:

- Im Betrieb als Host **muss** der LEON Konfigurationszugriffe auf andere PCI-Teilnehmer machen können, damit sie korrekt initialisiert werden und Adressbereiche zugewiesen bekommen. Im Betrieb als Nicht-Host ist dies nicht zwingend erforderlich, sollte und kann jedoch auch getestet werden, da Konfigurationszugriffe nötig sind, um andere PCI-Komponenten zu erkennen und zu identifizieren. Die Funktionalität der Konfigurationszugriffe ist Teil der Bridge.
- Im Betrieb als Nicht-Host **muss** die Bridge Adressräume auf dem PCI-Bus in die korrespondierenden Adressräume auf dem AHB-Bus übersetzen können, da nicht vorrausgesetzt werden kann, dass der Host die gleichen Adressindizes auf dem PCI-Bus vergibt, die AHB-intern festverdrahtet vorliegen. Beim Betrieb als Host ist dies nicht unbedingt notwendig, da in diesem Fall der LEON die Adressbereiche auf dem PCI-Bus festlegt, und dadurch eine Übereinstimmung zwischen PCI und AHB erreicht werden kann.
- Im Betrieb als Host müssen PCI-IRQs dem LEON signalisiert werden. Im Betrieb als Nicht-Host muss der LEON einen PCI-IRQ aktivieren können. Beides sind Funktionalitäten, die in die Bridge implementiert sind.

Bis auf die PCI-IRQ-Verarbeitung kann die Funktionalität durch den Betrieb als Nicht-Host getestet werden. Da zum Zeitpunkt der Diplomarbeit kein Hardwaresystem vorliegt, in dem der LEON als Host getestet werden kann, wird im Rahmen der Diplomarbeit auf die Tests mit dem LEON im Betrieb als Host verzichtet. Ein geeignetes Testsystem und Testsoftware ist in Zukunft zu entwickeln. Kapitel 4.2.1 beschreibt das verwendete Testsystem.

Hinsichtlich der zu testenden Komponenten lässt sich zwischen Komponenten unterscheiden, die für das Steuern bestimmter Schnittstellen verantwortlich sind, und solchen, die nur chipinterne LEON-Funktionen erfüllen. Beim Testen ersterer wird neben der LEON-Komponente auch die off-chip liegende Komponente getestet. Andererseits bedeutet dies, dass Fehler nicht zwangsläufig im LEON-Design liegen müssen. Anforderungen der externen Hardware müssen berücksichtigt werden. Kapitel 4.2.3 beschreibt die durchgeführten Tests, die sich mit Schnittstellen beschäftigten.
Auf Tests der internen LEON-Funktionen ist im Rahmen der Diplomarbeit nur teilweise eingegangen worden. Kapitel 4.2.4 beschreibt diese.

Fast alle Tests werden durch Abarbeitung von Programmen durch den LEON durchgeführt. Diese Programme werden durch einen, an den LEON angepassten, GNU-Crosscompiler erzeugt. Dieser angepasste Compiler wird von Gaisler Research zu Verfügung gestellt. Als Programmiersprache ist C und Assembler benutzt worden. Der Compiler erzeugt eine auf dem LEON ausführbare EXE-Datei.
Als Schnittstelle zum Datentransfer des Programmcodes wird UART 1 benutzt. Gaisler Research sieht dafür die Übertragung des Files im SREC-Format vor. Um aus dem EXE-File das korrespondierende SREC-File zu erzeugen, werden andere, durch Gaisler Reserach mitgelieferte Tools benutzt. Das SREC-File spezifiziert Befehlscode und Einsprungadresse des Programms.
Der VHDL-Code des unter 4.1.1 angeführten internen Boot PROMs kann auch durch die mitgelieferten Tools erzeugt werden. Der Code dieses Boot Programms initialisiert den LEON und liest Programme auf UART 1 ein (s. Kapitel 4.2.2).

4.2.1. Testumgebung

Wie unter 4.2 beschrieben, wird im Rahmen der Diplomarbeit der LEON im Betrieb als Nicht-Host in einem PCI-System getestet. Dieses PCI-System besteht aus dem PCI-Bus eines PCs, der für Einsteckkarten vorgesehen ist. Dieser PC wird im Folgenden mit PC 1 bezeichnet.
Ein zweiter PC (PC 2) steht zu Verfügung. Auf diesem werden BIT-Files erzeugt, Programme geschrieben und kompiliert.

Der LEON besitzt drei serielle Schnittstellen: UART 1, UART 2 und DCL (*DSU Communication Link*). Um das FPGA konfigurieren zu können, ist außerdem noch eine JTAG-Schnittstelle auf dem AVNET-Board vorhanden.

Tests ohne PCI-Benutzung und ohne UART 2-Benutzung wurden mit einem Aufbau wie in Abb. 4-1 dargestellt durchgeführt. Die AVNET-Karte steckt in einem PCI-Slot in PC 1.
Über die parallele Schnittstelle LPT 1 des PC 2 wird mittels der Xilinx Software IMPACT das BIT-File in das FPGA geladen. Dies ist nur notwendig, wenn die Konfigurationsdaten nicht im Konfigurations-Flash abgelegt sind (vgl. Kapitel 4.1). Der Wandler, der in der Abbildung dargestellt ist, generiert aus den Signalen, die IMPACT am parallelen Port erzeugt, die entsprechenden JTAG-Signale.
COM 1 von PC 2 ist mit UART 1 des LEON verbunden. Wie in Kapitel 4.2.2 beschrieben, werden die Testprogramme im SREC-Format über UART 1 eingelesen.
Der DCL des LEON ist mit COM 1 des PC 1 verbunden. Auf diesem PC läuft das Programm „DSU Monitor“ von Gaisler Research. Über dieses Kommandozeilen-Tool lässt sich leicht die Debug Unit des LEON bedienen.

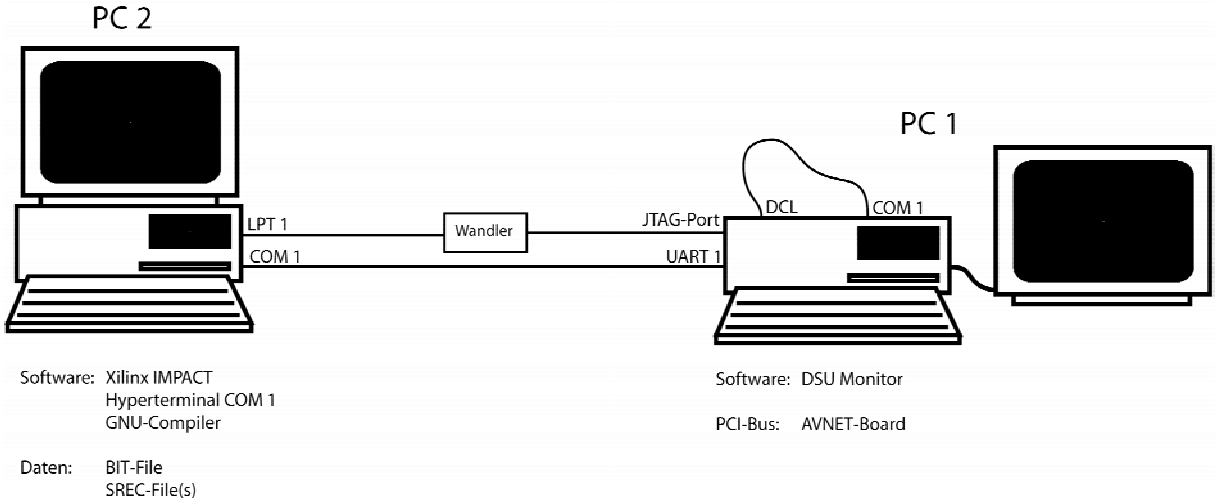


Abb. 4-1 Testaufbau 1

Soll UART 2 des LEON getestet werden, wird diese Schnittstelle an COM 2 von PC 2 angeschlossen (s. Abb. 4-2).

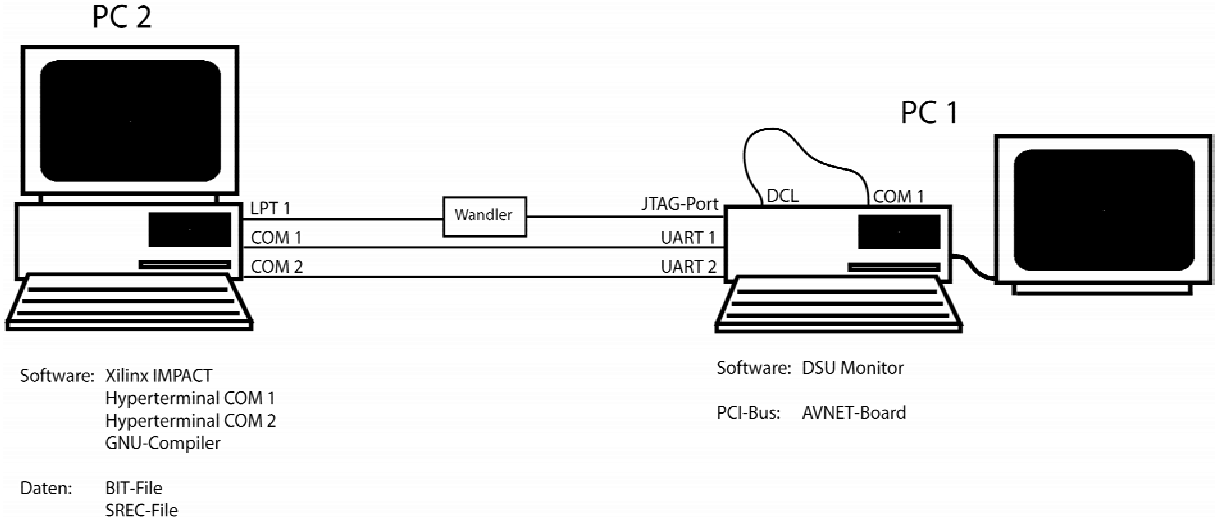


Abb. 4-2 Testaufbau 2 (UART 2 – Test)

Die Tests der PCI-Schnittstelle lassen sich in zwei Kategorien aufteilen (s. Kapitel 4.2.3.6):

- Der LEON ist der Initiator einer Übertragung. Diese Art der Übertragung kann durch Software für den LEON getestet werden, die solche Übertragungen programmiert. Als Adressat solcher Aktionen wird eine PCI-Einsteckkarte mit FPGA verwendet, die als Memory-Controller mit einem Speicherbereich von 4kB konfiguriert ist. Dies geschieht mithilfe eines IP-Cores der Firma Astrium. Die Ergebnisse können nach Abarbeitung des Testprogramms auch durch ein Software-Tool der Firma PLDA („PCI Tools“) überprüft werden. Dieses Tool erlaubt es, manuell Konfigurationsbereiche und Speicherbereiche von beliebigen PCI-Komponenten zu beschreiben und auszulesen. Den Aufbau zeigt Abb. 4-3.
- Die Bridge, bzw. ein dahinterliegender Adressbereich, wird von einem Teilnehmer des PCI-Busses adressiert. Diese Art von Übertragung kann nicht durch Software auf dem LEON getestet werden, da sie nicht vom LEON initiiert wird. Stattdessen wird dafür das Tool „PCI-Tools“ verwendet. Das Ergebnis solcher Übertragungen kann mit DSU

Monitor überprüft werden. Um Signale auf dem PCI-Bus überprüfen zu können, wird ein Bus Analyzer in Form einer PCI-Einsteckkarte benutzt. Dieser ist über eine RS232-Schnittstelle mit PC 2 verbunden (COM 2), auf dem die zugehörige Analyzer-Software („BusView“) läuft. Den Aufbau zeigt Abb. 4-4.

Anmerkung: Der verwendete PC 1 hat neben dem AVNET-Board nur noch einen freien PCI-Slot. Deshalb kann nicht der Memory-Controller gleichzeitig mit dem Bus Analyzer verwendet werden. Die Analyzer-Software läuft auf PC 2, um PCI-Signale besonders dann noch auswerten zu können, wenn PC 1 aufgrund von Fehlern im PCI-System abstürzt. Solche Fehler können Folge von falschen Konfigurationen oder fehlerhaften Signalen der PCI-Komponenten sein. Bei solchen Fehlern stürzt PC 1 ab und somit können die Aktionen, die den Absturz verursacht haben, nicht mehr mit PC 1 untersucht werden.

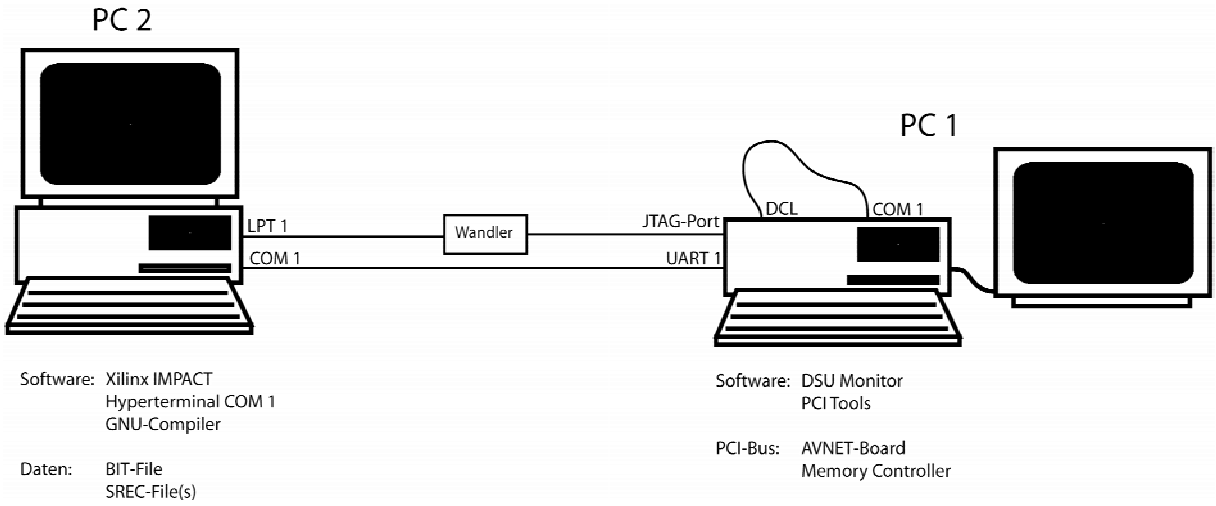


Abb. 4-3 Testaufbau 3 (PCI-Test: LEON als Initiator)

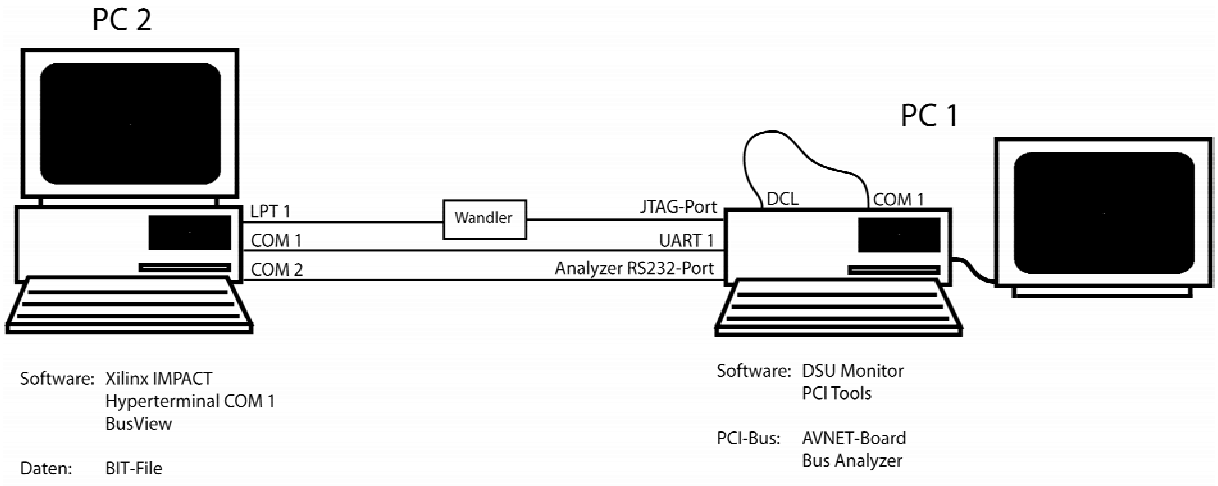


Abb. 4-4 Testaufbau 4 (PCI-Test: LEON wird adressiert)

4.2.2. Boot Loader

(Quellcode: Seite 117 ff)

Gaisler Research stellt einen Boot Loader als Quellcode zu Verfügung, der als internes Boot PROM implementiert werden kann. Dieses Programm hat die Aufgabe, die notwendigen Prozessorregister zu initialisieren und den Speicher zu konfigurieren. Anschließend wird eine Bootmessage über UART 1 gesendet und auf dieser Schnittstelle ein SREC-File erwartet, das im Anschluss an die Übertragung gestartet wird.
Im Laufe der Diplomarbeit stellte sich heraus, dass dieser Boot Loader nur für die Verwendung von SRAM geeignet ist. Da nur SDRAM auf dem AVNET-Board zu Verfügung steht, ist das Programm speziell für das AVNET-Board abgewandelt worden.
Zuerst werden *Processor State Register* (PSR) und *Window Invalid Mask Register* (WIM) gesetzt, so dass das aktuelle Register Fenster Nr. 7 ist. Traps werden nicht zugelassen, da, bis auf den Test der IRQs (s. Kapitel 4.2.4.1), keine Trap-Table programmiert wird. Um Fehlerquellen auszuschließen, werden sowohl der Befehls- als auch der Datencache deaktiviert. Beide UARTs werden aktiviert und auf 38 kbaud eingestellt. SRAM wird deaktiviert und SDRAM aktiviert und entsprechend den verwendeten Bausteinen die Parameter CAS, RAS, Organisation und Refresh Zyklus eingestellt. Anschließend wird der Stackpointer auf das Ende des SDRAM-Speicherbereiches gesetzt und folgende Boot-Message über UART 1 ausgegeben:

```
LEON-1:
> end initialization
```

Die Kommunikation mit UART 1 wird über ein Hyperterminal auf einem PC hergestellt. Das Senden von Zeichen über die serielle Schnittstelle wird durch ein Unterprogramm gemacht, das einen Zeichenstring zeichenweise in das UART1-Data-Register schreibt (s. Kapitel 2.2.1.6). Nach Senden der Boot Message wartet das Programm auf eine Executable in Form eines SREC-Files an UART 1. Ein SREC-File besteht aus Daten und aus einer Einsprungadresse. Anhand der ersten 2 Buchstaben kann zwischen diesen unterschieden werden. Diesen Buchstaben folgt eine Adresse, die entweder die Speicheradresse für die dahinterstehenden Daten angibt, oder die Startadresse für das Programm nach Speicherung der zuvor gesendeten Daten.
Der Boot Loader liest das SREC-File zeilenweise ein. Dies geschieht durch ein Unterprogramm, das solange UART1-Data- und UART1-Controll-Register abfragt, bis ein Zeilenumbruch empfangen wird. Die Zeile wird zwischengespeichert und anhand der Codierung entweder gar nicht benutzt (wenn das Format nicht SREC ist), die darin enthaltenen Daten an die auch darin enthaltene Adresse geschrieben oder die darin enthaltene Adresse als Einsprungadresse interpretiert. Danach wird die nächste Zeile eingelesen, außer es wurde eine Einsprungadresse empfangen. In diesem Fall wird eine Nachricht über den Empfang der Datei auf UART 1 ausgegeben:

```
transfer completed
```

Anschließend wird ein Unterprogrammaufruf zu der empfangenen Einsprungadresse gemacht. Das gesendete Programm wird abgearbeitet und nach dessen Beendigung wird in das Boot Loader Programm zurückgekehrt. Dieses gibt

```
program completed
```

auf UART 1 aus und beginnt wieder mit dem Einlesen eines SREC-Files auf UART 1. Abb. 4-5 stellt den Programmablauf graphisch dar. Ausgaben werden auf UART 1 gesendet.

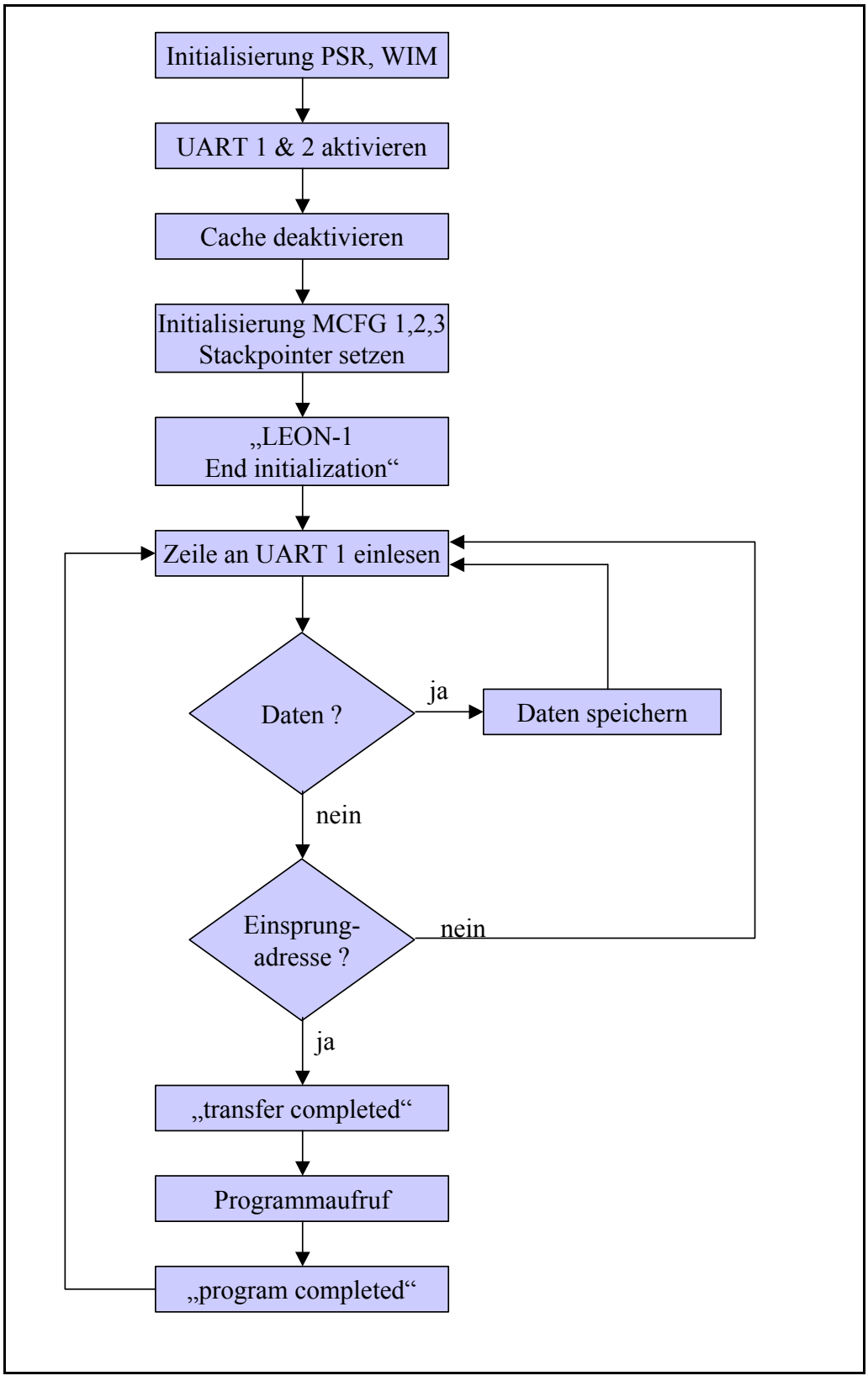


Abb. 4-5 Programmstruktur des Boot Loaders

4.2.3. Test der Schnittstellen und zugehöriger LEON-Funktionen

In der gewählten Konfiguration hat der LEON auf dem AVNET-Board folgende Schnittstellen, die er bedienen muss: DCL (DSU Communication Link), UART 1, UART 2, PIO, Flash-Memory, SDRAM, PCI. In Zukunft soll zusätzlich eine Ethernet-Schnittstelle zu Verfügung stehen. Dies zu realisieren, ist nicht Aufgabe der Diplomarbeit. Für den SDRAM wird kein spezielles Testprogramm entworfen, da alle hier beschriebenen Testprogramme im SDRAM abgelegt und von dort aus ausgeführt werden.
Im Laufe der Entwicklung der Testprogramme zeigte sich, dass nicht die Entwicklung der Testprogramme zeitaufwendig war, sondern das Verstehen der Schnittstellenkomponenten und die Einstellung von LEON-Registern bis Programme korrekt ausgeführt wurden. Im Rahmen der Diplomarbeit sollen grundlegenden Funktionen getestet werden. Einige Funktionen bleiben deshalb ungetestet. In den folgenden Kapiteln wird darauf gegebenenfalls hingewiesen.

4.2.3.1. Test DCL

Der DSU Communication Link (DCL) wird über eine RS232-Schnittstelle hergestellt, die mit einem PC verbunden ist. Der Test besteht darin, dass das von Gaisler Research mitgelieferte Programm DSU Monitor über diese Schnittstelle auf den Prozessor zugreifen kann. Die Benutzung der DSU war für die Entwicklung von Programmen im Laufe der Diplomarbeit zwingend erforderlich, um Fehler in diesen aufzuspüren.

4.2.3.2. Test UART 1

(Quellcode: Seite 120 ff)

Beim Booten gibt der LEON eine Boot Message auf UART 1 aus, die z.B. auf einem Hyperterminal eines PCs gelesen werden kann. Dies entspricht einer Übertragung von LEON zu PC. Die Richtung PC→LEON wird durch das Laden und korrekte Ausführen eines Programms wie unter 4.2.2 beschrieben getestet. Dazu wurde das berühmte „Hello World“ Programm benutzt. Dieses gibt fünfmal hintereinander

```
Hello World
```

auf UART 1 aus. Abb. 4-6 zeigt den Programmablauf. Die Ausgabe-Routine dieses Programms ist identisch mit der des Boot Loaders, ist jedoch im SDRAM abgelegt und wird dort adressiert. Zum Testen wird Testaufbau 1 (Abb. 4-1) benutzt. Im Rahmen dieses Tests wird nicht das Hardware-Handshaking getestet und nur Übertragungen bei 38 kbaud durchgeführt.

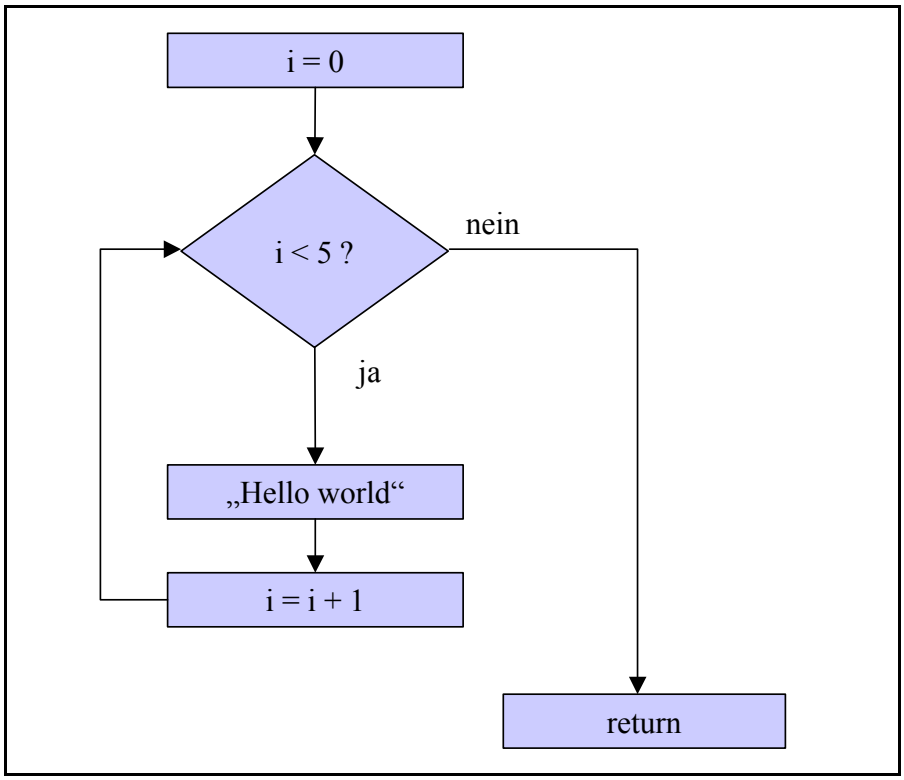


Abb. 4-6 Programmstruktur „Hello world“

4.2.3.3. Test UART 2

(Quellcode: Seite 122 ff und Seite 121)

Wie bei UART 1 (Kapitel 4.2.3.2) wird nicht das Hardware Handshake getestet und nur bei einer Übertragungsrate von 38 kbaud gearbeitet. Es wird Testaufbau 2 (Abb. 4-2) benutzt. Die UART 2 wird auf Seiten des PC durch ein Hyperterminal bedient. Es sind beide Richtungen zu testen: LEON→PC und PC→LEON.
Nach Start des Testprogramms wird zuerst vom LEON der Text

```
UART 2 listening
```

ausgegeben. Anschließend wird an UART 2 auf eine Textzeile beliebigen Formates mit maximal 256 Zeichen gewartet und diese eingelesen. Dafür wird ein Unterprogramm verwendet, das dem des Boot Loaders entspricht, mit der Ausnahme, dass an UART 2 gelesen wird. Um das Zeilenende zu erkennen, muss ein Zeilenumbruch mitgesendet werden.
Nachdem die Zeile eingelesen wurde wird

```
You sent:
```
gesendete Textzeile

ausgegeben und das Programm beendet. Alle Textausgaben werden auf UART 2 zum Testen und auf UART 1 zum Kontrollieren ausgegeben. Die Ausgaberoutine für UART 2 ist eine modifizierte Form der Routine für UART 1. Abb. 4-7 stellt die Programmstruktur graphisch dar.

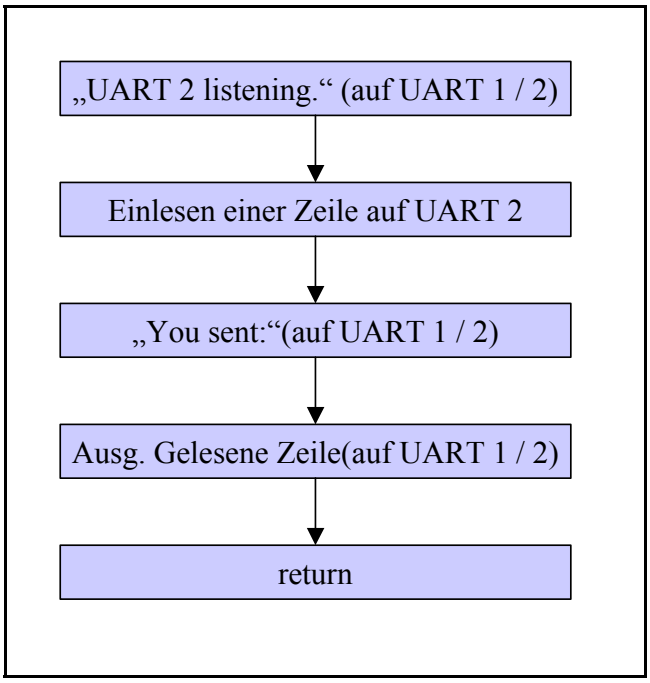


Abb. 4-7 Programmstruktur UART 2-Test

4.2.3.4. Test PIO

(Quellcode: Seite 123 und Seite 121)

Die parallele I/O-Schnittstelle (PIO) des LEONs besteht aus 32 Bit. Bei der gewählten Konfiguration des LEON und Verwendung des AVNET-Boards bleiben von den 32 nur 8 übrig, die noch keiner anderen Schnittstellenfunktion zugeordnet sind (vgl. Kapitel 2.2.1.7). Dies sind die Bits 7…0. Von diesen sind nach Durchlaufen des Place & Route Prozesses zwei an LEDs und sechs an Schalter angeschlossen. Zum Testen werden die in Tabelle 4-1 benannten Bits verwendet. Dabei müssen die beiden LEDs an Output-Ports, die Schalter an Input-Ports angeschlossen sein. Die Betriebsrichtung des jeweiligen Bits wird im PIO-Direction Register festgelegt.

LEON-Port	AVNET-Funktion
PIO(2)	LED 1
PIO(3)	LED 2
PIO(4)	Schalter 1
PIO(5)	Schalter 2
PIO(6)	Schalter 3
PIO(7)	Schalter 4

Tabelle 4-1 Belegung des PIO-Port

Das Testprogramm (Programmstruktur s. Abb. 4-8) fragt zuerst die aktuelle Konfiguration des PIO-Directions Register ab, um beim anschließenden Beschreiben die nicht vom Programm verwendeten Bits unverändert zu lassen. Nach Initialisierung der Hilfsvariable y (s.u.) wird in Abständen der PIO (PIO input/output Register) gelesen und ausgewertet.
Dabei ist vom Autor den PIO-Bits willkürliche Bedeutungen und Aktionen zugeordnet worden. Der Zustand von PIO(4) – d.h. Schalter 1 – wird PIO(2) zugewiesen. Wenn Schalter

1 auf HIGH steht, wird LED 1 zum Leuchten gebracht. Dieselbe Beziehung herrscht zwischen PIO(5) – d.h. Schalter 2 – und PIO(3) – d.h. LED 2.

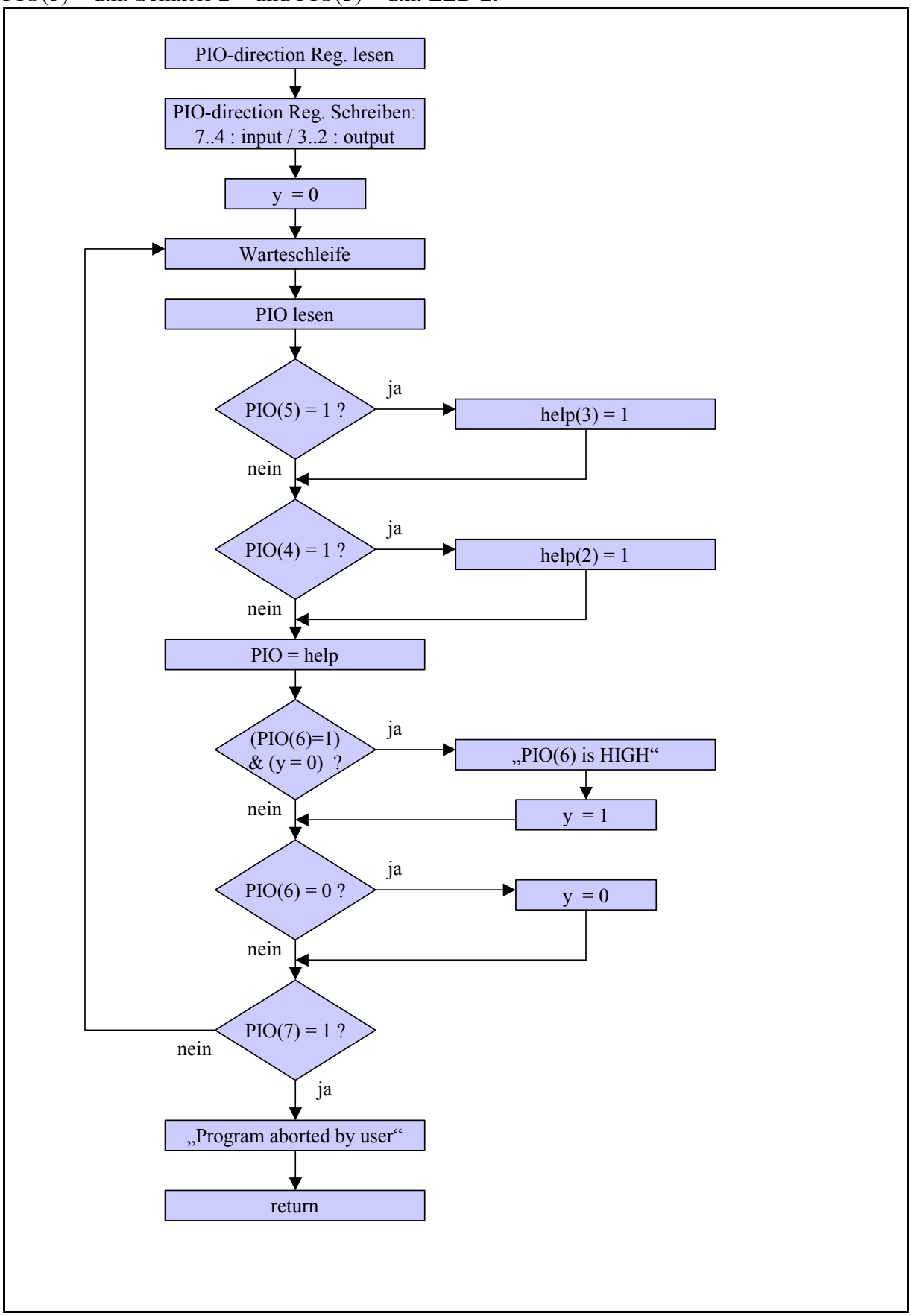


Abb. 4-8 Programmstruktur PIO-Test

Liegt an PIO(6) ein High-Level an, wird einmalig eine Ausgabe an UART 1 gesendet:

```
PIO(6) is HIGH
```

Einmalig bedeutet, dass nur beim Übergang von LOW auf HIGH von PIO(6) zwischen zwei Abfragen eine Ausgabe erfolgt. PIO(7) – d.h. Schalter 4 – erzwingt einen Programmabbruch mit der Ausgabe:

```
Program aborted by user
```

Setzen von PIO(2) und PIO(3) geschieht durch Schreiben des PIO-input/output Registers mit einer Eins an der jeweiligen Stelle. Die Software überprüft sequenziell PIO(4) und PIO(5). Zum Zwischenspeichern des Ergebnisses wird die Hilfsvariable *help* benutzt und anschließend in das PIO-Register geschrieben.
Um mehrmaliges Ausgeben eines Textes bei aktiven PIO(6) zu verhindern, wird die Variable *y* als Abfrage-Kriterium benutzt.

Das Programm testet nur die genannten sechs Bits des PIO in der beschriebenen Richtung. Andere Kombinationen oder die Funktionalität der IRQ-Erzeugung in Abhängigkeit von PIO-Eingangs-Bits werden nicht getestet. Der Test wird mit Testaufbau 1 durchgeführt (Abb. 4-1).

4.2.3.5. Test Flash-Memory

(Quellcode: Seite 124 ff und Seite 121)

Das AVNET-Board ist mit 4 Flash-Speichern ausgestattet, die in einer 64-Bit Konfiguration zusammengeschaltet sind. Der LEON unterstützt nur 32-Bit Speicherbreite. Deshalb sind nur zwei der Flashs benutzbar. Flash-Speicher behalten ihre Daten bei Stromausfall und sind programmierbar. Sie werden in der verwendeten Konfiguration als ROM Bank 1 an den LEON angeschlossen. D.h. sie sind ab Adresse 0x10000000 adressierbar.
Flash-Speicher kann wie SRAM gelesen werden, doch zum Beschreiben und Löschen (s.u.) des Flashs werden sog. Befehlssequenzen verwendet. Diese bestehen aus einer unterschiedlichen Anzahl von Schreibzugriffen (WE=’0’) auf ganz bestimmte Adressen mit festgelegten Daten. Diese verändern nicht den Inhalt des Flashs, sondern bedienen ein internes Steuerwerk. Die Abfolge dieser Sequenzen ist vorgeschrieben und eine Abweichung setzt das Flash wieder in den Lese-Modus. Die genaue Abfolge dieser Sequenzen ist dem Datenblatt zu entnehmen (AM29LV641 von AMD).

Die verwendeten Flashs können in zwei verschiedenen Modi programmiert werden. Soll nur ein Wort geschrieben werden, kann eine 4-schrittige Sequenz dafür benutzt werden. Sollen mehrere Daten geschrieben werden, sieht AMD einen sog. Bypass-Modus vor. In diesem kann durch eine 2-schrittige Sequenz ein Wort geschrieben werden. Um diesen Bypass-Modus zu aktivieren, muss zuerst eine sog. „unlock_bypass“-Sequenz (3 Schritte) ausgeführt werden. Um diesen Modus wieder ordnungsgemäß zu verlassen, muss eine sog. „unlock_bypass_reset“-Sequenz (2 Schritte) ausgeführt werden. Von dieser Methode wird im benutzten Programm Verwendung gemacht.
Der Bit-Inhalt des Flashs kann durch Programmieren nur von ‚1’ auf ‚0’ verändert werden. Ein Beschreiben eines Bits das ‚0’ ist mit einer ‚1’ hat keine Änderung zur Folge. Deshalb müssen vor Beschreiben eines Flashs alle Inhalte mit ‚1’ initialisiert werden. Dies wird als Löschen bezeichnet. Man hat die Auswahl zwischen Löschen des gesamten Flashs (ca. 3 min Dauer) und Löschen von einzelnen Sektoren (ca. 2 Sekunden Dauer). Die verwendeten Flashs

sind in 128 Sektoren unterteilt, die jeweils 64 kB zusammenhängenden Speicherbereich enthalten.

Das entwickelte Testprogramm ist in Abb. 4-9 graphisch dargestellt und besteht aus vier Teilen:

1) Nach Ausgabe von „Send SREC-File“ auf UART 1 wird ein File im SREC-Format eingelesen. Dies geschieht analog zum Einlesen einer Executable durch den Boot Loader. Es wird Anfangs-Adresse (*start_adr*) und End-Adresse (*end_adr*) gespeichert, um daraus die Größe der übertragenen Datei (*size*) zu ermitteln. Die Ausgabe „File received“ zeigt das Ende der Übertragung an.

2) Nach der Ausgabe „Starting erasing“ werden in Abhängigkeit der Anzahl der zu löschenden Sektoren entweder die benötigten Sektoren ab Sektor 0 oder das gesamte Flash gelöscht. Hintergrund dieser Unterscheidung ist der Zeitaufwand der für das Löschen benötigt wird. Ab ca. 100 zu löschenden Sektoren dauert das Löschen der einzelnen Sektoren länger als das Löschen des gesamten Flashs.

3) Um zu signalisieren, dass mit dem Programmieren begonnen wird, wird „Starting copying“ auf UART 1 ausgegeben. Danach wird eine „unlock_bypass“-Sequenz in Form eines Unterprogramms ausgeführt und das im SDRAM gespeicherte File wortweise in das Flash ab Adresse 0x10000000 kopiert. Die einzelnen Programmier-Sequenzen werden durch eine Routine realisiert. Dies ist nicht aus dem Programmablaufplan zu erkennen. Zum Abschluss wird mit einer „unlock_bypass_reset“-Sequenz das Flash wieder in den Lese-Modus versetzt.
   Anmerkung zur Abb. 4-9:
   Die Ausdruck *(*adresse*) zeigt auf den Inhalt der Adresse *adresse.*
   Zur Adressgenerierung wird der Zählindex *i* mit 4 multipliziert. Hintergrund dafür ist, das *size* aus programmiertechnischen Gründen so berechnet wird, dass es die Größe in 32-Bit Worten ausdrückt. Dies ist aus der Abbildung nicht zu erkennen.

4) Mit der Ausgabe „Verifying“ beginnt die Überprüfung der Kopie auf ihre Richtigkeit. Jedes Datum des Flashs wird mit dem korrespondierenden Inhalt des SDRAM verglichen. Bei Ungleichheit wird die Hilfsvariable *failure* gesetzt. Nachdem mit „Verifying completed:“ das Ende der Prüfung angezeigt worden ist, wird in Abhängigkeit von *failure* entweder „no errors“ oder „errors“ als Ergebnis der Prüfung ausgegeben und das Programm beendet.

Für diesen Test wird Testaufbau 1 (Abb. 4-1) benutzt. Es ist beim Kompilieren des SREC-Files, das als Datenfile zum Kopieren dient, darauf zu achten, dass der Inhalt nicht auf einen Adressbereich gelinkt wird, indem die Befehle des eigentlichen Testprogramms abgelegt sind. Das Programm beschreibt und überprüft nur den Adressbereich des Flashs, der durch das Datenfile ab Beginn des Flashadressbereiches belegt wird. Es wird also nicht der gesamte Speicherbereich des Flashs überprüft.

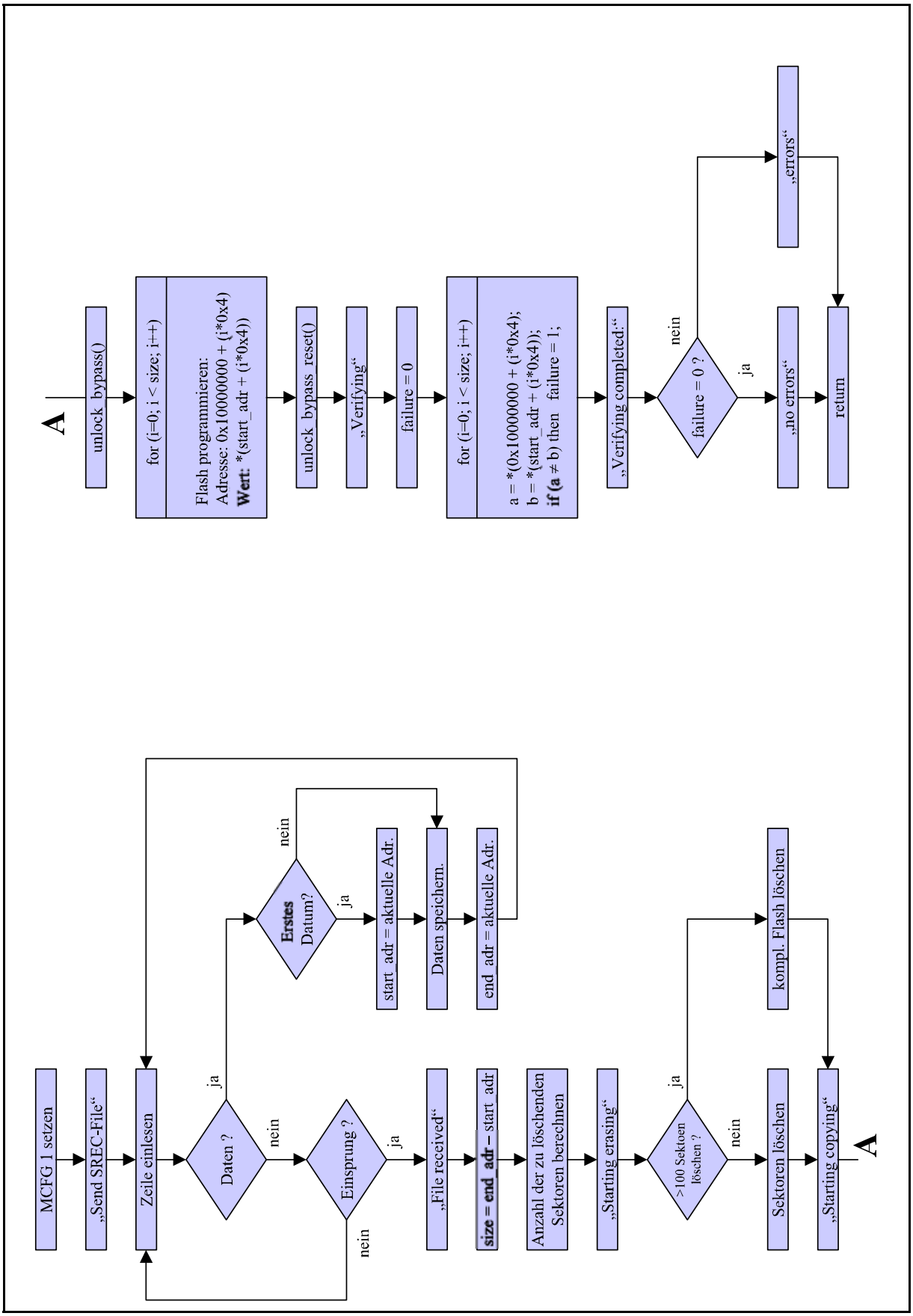


Abb. 4-9 Programmstruktur Flash-Test

4.2.3.6. Test PCI

(Quellcode: Seite 127 ff und Seite 121)

Beim Testen der PCI-Schnittstelle muss zwischen zwei verschiedenen Szenarien unterschieden werden. Im einen Fall ist der LEON der Initiator einer Übertragung, im anderen Fall ein anderer PCI-Teilnehmer.

LEON als Initiator

Für den Test der PCI-Schnittstelle mit dem LEON als Initiator wird der Testaufbau 3 (Abb. 4-3) verwendet. Da in diesem Test der LEON Aktionen initiiert, kann der Test durch eine Software ausgeführt werden. Es werden Lese- und Schreibzugriffe auf Konfigurationsbereich und Speicherbereich eines anderen PCI-Teilnehmers getestet. Die Verwendung des PCI-IRQs wird nicht getestet. Der adressierte PCI-Teilnehmer ist eine als Memory-Controller deklarierte PCI-Einsteckkarte, die 4kB SRAM verwaltet.
Das Testprogramm durchsucht zuerst den PCI-Bus nach einem Memory-Controller. Dies geschieht über Lesezugriffe auf den Konfigurationsbereich. Wird die gewünschte Komponente gefunden, wird dies über UART 1 mitgeteilt, die Adresse auf dem PCI-Bus gelesen und wenn nötig durch einen Schreibzugriff auf den Konfigurationsbereich des Memory-Controllers die Möglichkeit des Zugriffs auf seinen Speicher aktiviert. Wird kein Memory-Controller gefunden, wird dies auch an UART 1 mitgeteilt und das Programm abgebrochen.
Bei Erkennung des Controllers wird im Anschluss ähnlich des Flash-Tests (s. Kapitel 4.2.3.5) verfahren:

1) Einlesen eines SREC-File an UART 1 und Speichern im SDRAM
2) Kopieren der empfangenen Daten in den Memory-Controller
3) Auslesen des Memory-Controllers und Vergleich mit den Daten im SDRAM

Im Unterschied zum Flash-Test kann auf den Memory-Controller zugegriffen werden wie auf normalen SRAM. D.h., dass eine Übertragung auf dem AHB-Bus ein Wort liest oder schreibt. Löschen ist nicht notwendig, da der adressierte Speicher SRAM ist.

Anmerkung: Im Laufe der Entwicklung dieses Testprogramms stellte sich heraus, dass der LEON immer in Error Mode geht, wenn eine Übertragung auf dem AHB-Bus mit Signalisierung eines Fehlers (oder Retry) beendet wird. Dies geschieht z.B., wenn eine Übertragung vom LEON auf dem PCI-Bus initiiert wird, aber ohne Datenübertragung beendet wird.
Da der LEON keine Kenntnis über die an den PCI-Bus angeschlossenen Komponenten hat (max. 16 am direkt angrenzenden Bus), versucht er, jede mögliche Komponente beim Durchsuchen nach dem Memory-Controller zu adressieren. Da nicht zwangsläufig alle theoretisch möglichen PCI-Komponentenplätze belegt sind, können Zugriffe auf nicht vorhandene Komponenten eintreten. Diese werden selbstständig durch die AMBA-PCI-Bridge nach den PCI-Konventionen abgebrochen. Dies wird an das AHB-Bus-Interface weitergegeben, das darauf einen AHB-Fehler signalisiert.
Um solche Zugriffe trotzdem ausführen zu können, ohne den LEON zu stoppen (Error Mode), ist ein zusätzliches Register in die Bridge implementiert worden, mithilfe dessen die Erzeugung von Fehlern oder Retry auf dem AHB-Bus unterdrückt werden kann. Im gleichen Register werden diese Fehlermeldungen jedoch gespeichert. Dieses Register kann über das

APB-Interface vom LEON aus gelesen und geschrieben werden. Dadurch kann über Software die Korrektheit einer Übertragung überprüft werden.

PCI-Komponente als Initiator

Soll die PCI-Schnittstelle mit einer PCI-Komponente als Initiator und dem LEON als adressierter Einheit getestet werden, kann dies nicht durch Abarbeitung eines Programms auf dem LEON gemacht werden. Stattdessen wird die Software „PCI-Tools“ der Firma PLDA benutzt. Über eine graphische Oberfläche (GUI) lassen sich hiermit gezielt Speicherinhalte der PCI-Komponenten verändern. Auch längere Sequenzen von Übertragungen können benutzt werden. Dies sind z.B. Schreiben einer Rampe in den Speicher oder Löschen des Speichers. Das Tool liest im Anschluss an jede Aktion den auf dem GUI sichtbaren Teil des Speichers aus und überprüft auf Korrektheit. Es wird der Testaufbau 4 (Abb. 4-4) verwendet.

Die Tests konnten nicht zufriedenstellend durchgeführt werden. Zwar wurden einzelne Operationen korrekt ausgeführt, doch zeigten sich bei längeren Sequenzen in ca. 20 % der Versuche Fehler. Diese Fehler bestanden entweder aus nicht geschriebenen Daten, aus versetzt geschriebenen Datensequenzen oder aus an die falsche Adresse geschrieben Daten. Ein Zusammenhang zwischen Auftreten von Fehlern und Übertragungssequenz konnte nicht festgestellt werden.
Um die Fehlerursache einzugrenzen, wurde gleichzeitig der PCI-Bus mit einem Bus-Analyzer gescannt und der AHB-Bus mithilfe des DSU-Trace-Buffers. Die Ergebnisse zeigen, dass die Aktionen auf dem PCI-Bus korrekt durchgeführt und von der Bridge angenommen werden. Im Falle der fehlerhaften Sequenzen sind die Übertragungen auf dem AHB-Bus falsch, d.h. Adresse und/oder Daten sind nicht korrekt. Die Ursache für die Fehler ist also die Bridge oder das Zusammenspiel zwischen Bridge und AHB-Bus.
Das komplette System mit AMBA-PCI-Bridge ist sowohl funktional als auch zeitlich simuliert worden. Die Fehler konnten jedoch nicht reproduziert werden. Aufgrund der Beliebigkeit der Fehler und eines Indiz – die Übereinstimmung bridge-interner Registerwerte, die nicht zum Senden benutzt werden, aber vor der Übertragung an den AHB-Schnittstellen anliegen, mit einer bestimmten fehlerhaften Übertragung auf dem AHB-Bus – lässt vermuten, dass die Fehler auf Timing-Probleme zurückzuführen sind.
Eine spätere genaue Fehleranalyse muss die tatsächliche(n) Fehlerquelle(n) aufdecken, um die Verbesserung des Designs zu ermöglichen. Dies kann nicht mehr im Rahmen der Diplomarbeit geschehen.

4.2.4. Test der internen LEON-Funktionen

Im Rahmen der Diplomarbeit wurden nur einige LEON-interne Funktionen getestet: IRQ-Controller, Timer 1, Timer 2. Alle anderen Funktionstests müssen in Zukunft durchgeführt werden.

4.2.4.1. IRQ-Controller / Timer 1 / Timer 2

(Quellcode: Seite 129 ff und Seite 121)

Timer 1 und Timer 2 können IRQs auslösen. Deshalb ist der Test dieser Komponenten mit dem Test des IRQ-Controllers zusammengelegt worden. Für den Test wird Testaufbau 1

(Abb. 4-1) verwendet. Es wird die in Kapitel 4.2.3.4 beschriebene Belegung der PIO-Bits (7..4) benutzt, um bestimmte Aktionen zu aktivieren:

PIO(4): Setzt den IRQ 13 im IRQ-Force-Register (s. Kapitel 2.2.1.4) manuell
PIO(5): Aktiviert Timer 1
PIO(6): Aktiviert Timer 2
PIO(7): Bricht das Programm ab

Abb. 4-10 zeigt die Programmstruktur. Zur Kompilierung wird auch eine Trap Table und Routinen für die Interrupt-Behandlung eingebunden, die nicht in der Abbildung dargestellt sind. Jede Interrupt-Behandlung besteht aus einer Ausgabe auf UART 1:

„`IRQ13 activated`“ bzw. „`Timer1-IRQ`“ bzw, „`Timer2-IRQ`“

Das Programm konfiguriert zuerst die PIO-Schnittstelle für seine Bedürfnisse. Anschließend werden die Timer-Register so gesetzt, dass Timer 1 in ca. 3 Sekunden Abständen IRQs erzeugt und Timer 2 in ca. 6 Sekunden Abständen. Mögliche noch ausstehende IRQs im IRQ-Pending –Register des IRQ-Controllers werden gelöscht. Die IRQs 8, 9 (Timer 1, Timer 2) und 13 werden demaskiert bevor Traps zugelassen werden. Danach wird der PIO zyklisch abgefragt und ausgewertet. Die Hilfsvariablen *x*, *y*, *z* verhindern, dass entweder kontinuierlich IRQ13 gesetzt wird oder Timer 1 / 2 ständig neu aktiviert werden, was im vorliegenden Programm ein erneutes Laden des Zählers und damit keine IRQ-Generierung zur Folge hätte. Vor dem Abbruch des Programms werden Traps wieder deaktiviert.

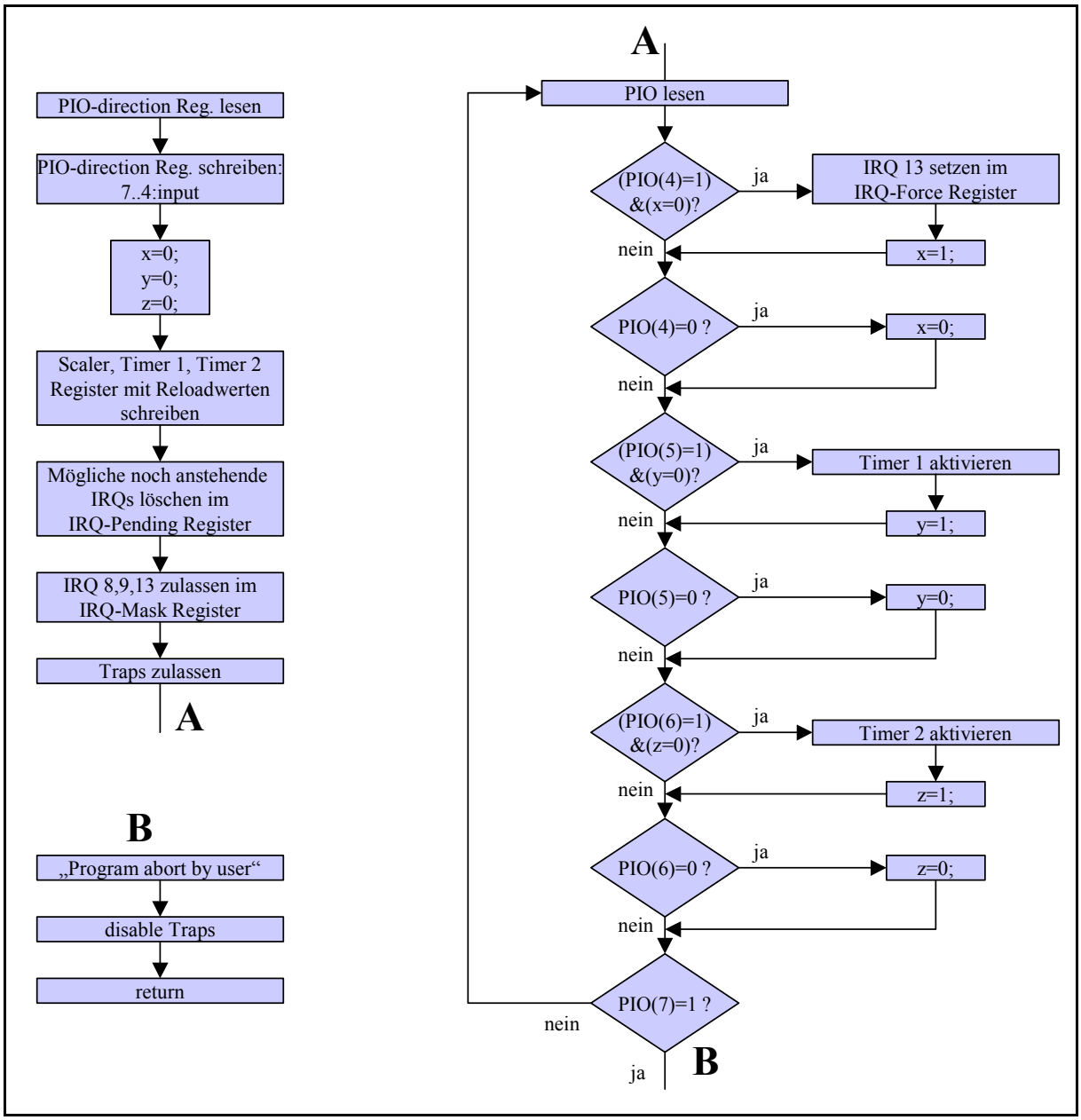


Abb. 4-10 Programmstruktur für Test des IRQ-Controller / Timer 1 / Timer 2

4.3. Zukünftige Tests

Im Rahmen der Diplomarbeit sind nur Tests durchgeführt worden, die allgemeine Funktionen testen und als erster Beweis für die Funktionsfähigkeit des Designs dienen. Tests, die für kommerzielle Zwecke durchgeführt werden müssen, sind noch zu entwickeln. Bevor diese Tests jedoch durchgeführt werden, muss zuerst eine Fehleranalyse und Fehlerbehebung bezüglich der PCI-AMBA-Bridge gemacht werden, da die durchgeführten Tests kein zufriedenstellendes Ergebnis zeigten (vgl. Kapitel 4.2.3.6).
Im Folgenden sind Testziele beschrieben, die aus Sicht des Autors zwingend erforderlich sind, um eine umfassende Funktionsfähigkeit des Designs gewährleisten zu können.

- Um die Benutzung einer Ethernet-Schnittstelle zu ermöglichen, muss ein Ethernet-MAC in das LEON-Design integriert und getestet werden. Ein solcher MAC liegt als VHDL-Core vor. Es sind Datenströme mit anderen Ethernet-Komponenten auszutauschen. Dies kann z.B. ein PC sein.
- Die PCI-IRQ-Verarbeitung und Erzeugung muss getestet werden. Bei Erzeugung eines IRQ muss zumindest das Aktivieren auf dem PCI-Bus sichtbar gemacht werden. Besser wäre eine IRQ-Service-Routine, die z.B. auf dem Host-PC läuft. Das Verarbeiten von PCI-IRQs durch den LEON geschieht in einem PCI-System, in dem der LEON der Host ist.
- Alle internen LEON-Funktionen müssen getestet werden. Dies sind z.B. Cache, Multiplizierer und Dividierer.
- Der LEON muss als Host getestet werden. Dafür ist Hardware zu entwickeln oder zu benutzen, in der auch die Funktionalität eines PCI-Arbiters getestet werden kann. Diese Komponente wird durch Gaisler Research zu Verfügung gestellt, konnte aber nicht getestet werden, da pro PCI-Komponenten ein *Request*- und ein *Grant*-Signal an den Arbiter geführt werden müssen. Dies ist durch einen PC-PCI-Steckplatz nicht gewährleistet.

5. Implementierung und Test des LEON auf PMC-Board

Die ursprüngliche Planung der Diplomarbeit umfasste die Inbetriebnahme des entwickelten PMC-Boards und Tests des LEONs darauf. Da zum 31.07.03 das Layout des Boards noch nicht abgeschlossen ist und die Produktion, die zum Teil an Fremdfirmen vergeben wird, nach dem Layout noch ca. 6 Wochen benötigt, muss darauf verzichtet werden.
Die Testverfahren für das Board sind vorerst schon festgelegt worden. Nach Absprache mit dem zuständigen Betreuer sollen die für die Tests notwendigen Entwicklungen erst in Zukunft durchgeführt werden, um eventuelle Änderungen im Design oder in der Zielsetzung der Tests miteinbeziehen zu können.

Vor Inbetriebnahme des Boards müssen die elektrischen Verbindungen überprüft werden. Dies geschieht durch die Abteilung, die die Bestückung der Leiterplatte ausführt. Um die Funktionsfähigkeit der verschiedenen Bausteine zu überprüfen, sind Teststrategien festgelegt worden. Diese sind in Kapitel 5.1 beschrieben.
Die Tests des LEON und seiner Komponenten entsprechen den in Kapitel 4beschriebenen. Bei der Implementierung des LEONs mit internem Boot PROM ist darauf zu achten, den von Gaisler Research zu Verfügung gestellten Boot Loader zu verwenden, da das PMC-Board nicht mit SDRAM sondern mit SRAM bestückt ist. Das Pinout des FPGAs ist dem Board anzupassen.
Als Testumgebung wird das in Kapitel 2.1.2 beschriebene PMC-Trägerboard verwendet, das an einem VME-Bus angeschlossen wird. Das Board liefert die Betriebsspannung und die PCI-Bus-Anbindung. Es bietet außerdem noch einen zweiten PMC-Steckplatz, der mit einem noch nicht spezifizierten Board bestückt wird, um auf dieses PCI-Zugriffe vom LEON aus zu initiieren. Um den LEON über den PCI-Bus zu adressieren, kann entweder dieses zweite PMC-Board verwendet werden oder ein Teilnehmer des VME-Bus, an den das Trägerboard angeschlossen ist.

5.1. Test der Bauteile

Die einzelnen Bauteile sind auf ihre Funktionsfähigkeit hin zu überprüfen. Da alle als Peripherie für den LEON entworfen worden sind und dieser im FPGA untergebracht wird, werden sie durch FPGA-Designs getestet. Welche Baugruppen dies sind wird im Folgenden beschrieben:

- RS232-Treiber: Diese Bausteine beinhalten keine logische Funktionalität sondern sind Pegelwandler. Sie werden im Rahmen des Tests des LEON mitgetestet.
- Ethernet-PHY: Als Testdesign wird ein vorhandener IP-Core verwendet. Dieser implementiert einen Ethernet-MAC.
- Flash-Memory: Der Flash-Speicher wird durch Abarbeitung eines Programms auf dem LEON getestet.
- SRAM: Der SRAM wird an einen vorhandenen IP-Core eines PCI-Memory-Controllers angehängt und über den PCI-Bus getestet. Dafür ist der IP-Core dahingehend zu modifizieren, dass nicht der Speicher auf dem FPGA angesteuert wird, sondern der auf dem PMC-Board vorhandene.
- LEDs, Schalter, Taster: Es ist ein einfaches VHDL-Testdesign zu entwickeln, das alle Schalter und Taster abfragt und alle LEDs ansteuert.

- PMC-Schnittstelle: Die PCI-Schnittstelle des PMC-Boards muss getestet werden, um sicherzustellen, dass eine Kommunikation mit dem PCI-Bus möglich ist. Zu lange Laufzeiten der PCI-Signale auf dem PMC-Board könnten zu Fehlern führen. Als Test-Design wird ein vorhandener IP-Core verwendet, der sowohl Übertragungen auf dem PCI-Bus initiieren als auch Ziel davon sein kann. Dadurch können alle Signale des PCI-Busses getestet werden.

6. Fazit und Ausblicke

Zielsetzung der vorliegenden Diplomarbeit waren drei Hauptschwerpunkte:

- Entwicklung eines PMC-Boards, auf dem das LEON-Design implementiert werden kann.
- Implementierung und Test der grundlegenden Funktionen des LEONs auf einem Evaluations-Board.
- Inbetriebnahme des entwickelten PMC-Boards, Implementierung und Test des LEONs darauf.

Außerdem war eine Aufgabe der Diplomarbeit, Interrupts auf dem PCI-Bus in einer geeigneten Form mit dem LEON zu verbinden, so dass dieser sowohl als Host eines Systems auf PCI-Interrupts reagieren kann, als auch als Nicht-Host Interrupts auslösen kann. Diese Aufgabe nahm den kleinsten Teil der aufgewendeten Zeit in Anspruch. Die entwickelte Funktionalität ist bisher nur funktional simuliert.

Für die Entwicklung des PMC-Boards wurde die meiste Zeit aufgewendet. Es zeigte sich während des Entwicklungsprozess, dass nicht alle Ansprüche, die von verschiedenen Seiten an das entwickelte Board gestellt wurden, zu erfüllen waren. So war ein Anspruch, möglichst für die Raumfahrt qualifizierte Bauteile zu verwenden, ein anderer, Kosten zu sparen, ein dritter, für Testzwecke möglichst große funktionale Flexibilität zu gewährleisten, und ein vierter, die räumlich einengenden Kriterien der PMC-Spezifikation zu erfüllen. Folge dieser verschiedenen Kriterien war ein erheblicher Zeitaufwand, der für Bauteilauswahl und Rücksprache mit Mitarbeitern aufgewendet wurde.

Da das entwickelte Board ein Testboard ist, sind die Ansprüche bezüglich der funktionalen Flexibilität vorrangig behandelt worden. Nach Möglichkeit wurden dabei Bauteile verwendet, die qualifizierten Bauteilen äquivalent sind, um späteren Redesignaufwand zu minimieren. Aus Kostengründen wurden - wenn möglich – Bauteile, die im Lager vorhanden waren, verwendet. Die PMC-Spezifikation wurde, wie in Kapitel 3.1.2.4 beschrieben, verletzt.

Für eine kommerzielle Version müssen die PMC-Spezifikationen aber eingehalten und u.U. mehr Speicher zu Verfügung gestellt werden. Ob Änderungen, die für eine kommerzielle Version gemacht werden müssen, akzeptabel sind, ist nicht im Rahmen der Diplomarbeit behandelt worden. Dies hängt davon ab, ob der vorgesehene Speicher ausreicht und welche Schnittstellentypen vorhanden sein müssen. Wird der benötigte Platz- und Raumbedarf zu groß, muss ein anderer Boardtyp eingesetzt werden

Da das PMC-Board zum Ende der Diplomarbeit noch nicht layoutet ist und dementsprechend noch mehrere Wochen bis zur Fertigstellung benötigt, konnte es nicht in Betrieb genommen werden. Nach Maßgabe des zuständigen Mitarbeiters sind nur Richtlinien für die Tests der Bauteilgruppen auf dem Board festgelegt worden. Eine Entwicklung oder Modifizierung der für diese Tests benötigten VHDL-Designs muss in Zukunft gemacht werden. Hintergrund dafür ist, dass eventuelle Änderungen am PMC-Design nicht ausgeschlossen sind, und diesen nach Abschluss des Layouts und Fertigungsbeginn besser Rechnung getragen werden kann.

Die Implementierung des LEONs auf dem Evaluations-Board konnte erfolgreich durchgeführt werden. Es wurde dazu eine Konfiguration des LEON verwendet, die nicht alle optionalen Komponenten implementiert. Zur Anbindung der PCI-AMBA-Bridge mussten nur wenige

Änderungen an deren Source-Code gemacht werden. Allerdings zeigte sich im Verlauf der Tests, dass die Bridge zusätzliche Funktionalität implementiert bekommen musste, um Fehlermeldungen auf dem AHB-Bus zu unterdrücken, die den Prozessor in Error-Modus setzten und damit eine weitere Abarbeitung von Programmen unmöglich machten. Die Erfahrungen durch das Einbinden der Bridge zeigen, dass ein Anbinden anderer Komponenten an den AHB-Bus möglich ist und mit geringem Aufwand bewerkstelligt werden kann. Dazu muss lediglich die Komponente instantiiert werden, mit vorhandenen AHB-Signalen einerseits und neu hinzuzufügenden Ports des LEON andererseits verbunden werden. Außerdem muss in den Konfigurationseinstellungen des LEON der veränderten Anzahl an AHB-Teilnehmern Rechnung getragen werden. Eine solche zusätzliche AHB-Komponente ist z.B. ein Ethernet MAC. Wird dieser implementiert, kann eine Ethernet-Schnittstelle angesteuert werden. Dies ist für das PMC-Board gefordert.

Die Hauptarbeit bei der Entwicklung der Testsoftware für den LEON besteht im Verständnis der Funktionsweise des LEONs, dessen Register, sowie eventuell off-chip angeschlossener Bauteile. Um zum gewünschten Ergebnis zu kommen, muss das Handling und Zusammenspiel aller Komponenten stimmen. Um Softwareentwicklern größere Komfortabilität zu gewährleisten, ist es für die Zukunft sinnvoll, Programmroutinen zu entwickeln, auf die zurückgegriffen werden kann, ohne über die Kenntnis und die Funktionsweise einzelner Register bzw. Registerbits verfügen zu müssen.

Die im Rahmen der Diplomarbeit durchgeführten Tests beschäftigen sich mit den grundlegenden Funktionen des LEONs. Der Großteil davon besteht aus Tests der Schnittstellen des LEONs nach draußen: PCI, SDRAM, Flash-Memory, serielle Schnittstellen und PIO. Als interne Funktionen sind der Interrupt-Controller und die beiden Timer getestet worden. Der Test der PCI-Schnittstelle verlief nicht erfolgreich. Vermutlich sind Timing-Probleme innerhalb der Bridge oder im Zusammenspiel zwischen Bridge und AHB-Bus die Ursache für Fehler, die sowohl im Testsystem als auch in der Timing-Simulation nicht reproduzierbar sind. Eine genaue Fehleranalyse und Fehlerbehebung ist in Zukunft zwingend erforderlich, um den LEON mit der verwendeten Bridge an den PCI-Bus anschließen zu können.

Alle anderen Tests verliefen erfolgreich. Da die vorhandenen Tests aber nur die grundlegenden Funktionen überprüfen und nicht alle Funktionalitäten und Eventualitäten abdecken, müssen in Zukunft umfangreichere Tests entwickelt werden. Diese müssen z.B. auch die noch nicht getesteten internen LEON-Funktionen wie Cache, Multiplizierer und Dividierer testen. Da diese Komponenten in verschiedenen Konfigurationen implementiert werden können, müssen verschieden konfigurierte Designs getestet werden.

Alle Tests sind für ein zukünftiges PMC-Board verwendbar, jedoch muss der veränderten Hardwareumgebung Rechnung getragen werden: Satt SDRAM wird SRAM benutzt und statt Flash-Memory mit 16-Bit Wortbreite wird solcher mit 8-Bit-Wortbreite verwendet. Dieses hat Änderungen in der benötigten Einstellung von Speicherkonfigurationsregistern bzw. Programmiersequenzen der Flashs zur Folge, die durch Boot Loader bzw. Testprogramme bedient werden. Für die Bedienung der Ethernet-Schnittstelle ist ein Ethernet-MAC zu implementieren und entsprechende Testsoftware zu entwickeln.

Unter der Bedingung, dass die Fehler bei PCI-Übertragungen behoben werden und umfassende Tests des gesamten LEON-Designs erfolgreich durchgeführt werden, kann das vorliegende Design nach Meinung des Autors als Prozessor für Applikationen verwendet werden.

A-1 Quellenverzeichnis

Den einzelnen Quellen ist – wenn vorhanden – ein Kürzel für die Internet-Quelle beigefügt. Unter Umständen ist das Dokument nicht direkt zu beziehen. In einigen Fällen ist ein Mitglieds-Account Vorraussetzung für den Bezug der Daten. In anderen Fällen muss die Quelle käuflich erworben werden.

Literatur

[L1] Shanley, Anderson: *PCI System Architecture*, Addison-Wesley
[L2] Tietze, Schenk: *Halbleiter-Schaltungstechnik*, 10.Auflage, Springer-Verlag
[L3] Becke, Haseloff: *Das TTL-Kochbuch*, Texas Instruments
[L4] Röben: *Analyse und Anfertigung von Designmethoden zur Erhöhung der SEU-Resistenz in Xilinx Virtex FPGAs*, Diplomarbeit, FH-Oldenburg, Ostfriesland, Wilhelmshaven
[L5] PCI SIG: *PCI Local Bus Specification*, Revision 2.2 [W2]
[L6] PCI SIG: *PCI-to-PCI Bridge Specification*, Revision 1.0 [W2]
[L7] IEEE: *Draft Standard for a Common Mezzanine Card Family: CMC* , Draft 2.4a [W3]
[L8] IEEE: *Draft Standard Physical and Enviromental Layers für PCI Mezzanine Cards: PMC* , Draft 2.4 [W3]
[L9] Intel Corporation: *PC SDRAM Specification* [W4]
[L10] Gaisler Research: *The LEON-2 Processor User's Manual*,Version 1.0.10 [W5]
[L11] Gaisler Research: *DSU Monitor User's Manual*,Version 1.0.5 [W5]
[L12] Gaisler Research:*The LEON/ERC32 GNU Cross-Compiler System*,Version 1.1.5 [W5]
[L13] Kasprzyk: *Floating Point Unit Digital IC project 2001* [W5]
[L14] SPARC International Inc.: *The SPARC Architecture Manual*, Version 8 [W6]
[L15] Mader: *PCI-Core Master/Target – PCI-AMBA-Bridge*
[L16] ARM Limited: *AMBA Specification*, Revision 2.0 [W7]
[L17] ARC International: *VMAC Reference*, Version 2002.03 Revision 5

Datenblätter

[D1] Kingbright: AP2012 Series [W8]
[D2] Micropac Industries, Inc.: 1N6609/1N6610/1N6611
[D3] Jauch Quartz GmbH: Oscillator VX3 3.3 V [W9]
[D4] Intel Corporation Inc.: Intel LXT971A [W4]
[D5] Linear Technology: LT1130A/LT1140A Series [W10]
[D6] Maxim Integrated Products: MAX706P/R/S/T, MAX708R/S/T [W11]
[D7] Linear Technology: LT1083/LT1084/LT1085 [W10]
[D8] Advanced Micro Devices: Am29F016D [W12]
[D9] NEC Electronics Corporation: µPD434008A [W13]
[D10] Xilinx, Inc.: XC95144XL [W1]
[D11] Xilinx, Inc.: XC18V00 [W1]
[D12] Xilinx, Inc.: Virtex 2.5V FPGAs [W1]
[D13] Xilinx, Inc.: Virtex-E 1.8V FPGAs [W1]
[D14] Xilinx, Inc.: QPro Virtex 2.5V Radiation Hardened FPGAs [W1]
[D15] Xilinx, Inc.: QPro XQ18V04 (XQR18V04) QML [W1]
[D16] Electronic System Design GmbH: VME-PMC-Caddy

[D17] Tundra Semiconductor Corporation: Universe [W14]
[D18] Avnet, Inc.: Virtex-E Development Kit, XCV1000E FPGA, User's Manual
[D19] Avnet, Inc.: Virtex-E Development Kit, XCV1000E FPGA, Schematics
[D20] Avnet, Inc.: Virtex-E Development Kit, XCV1000E FPGA, Bill of Materials

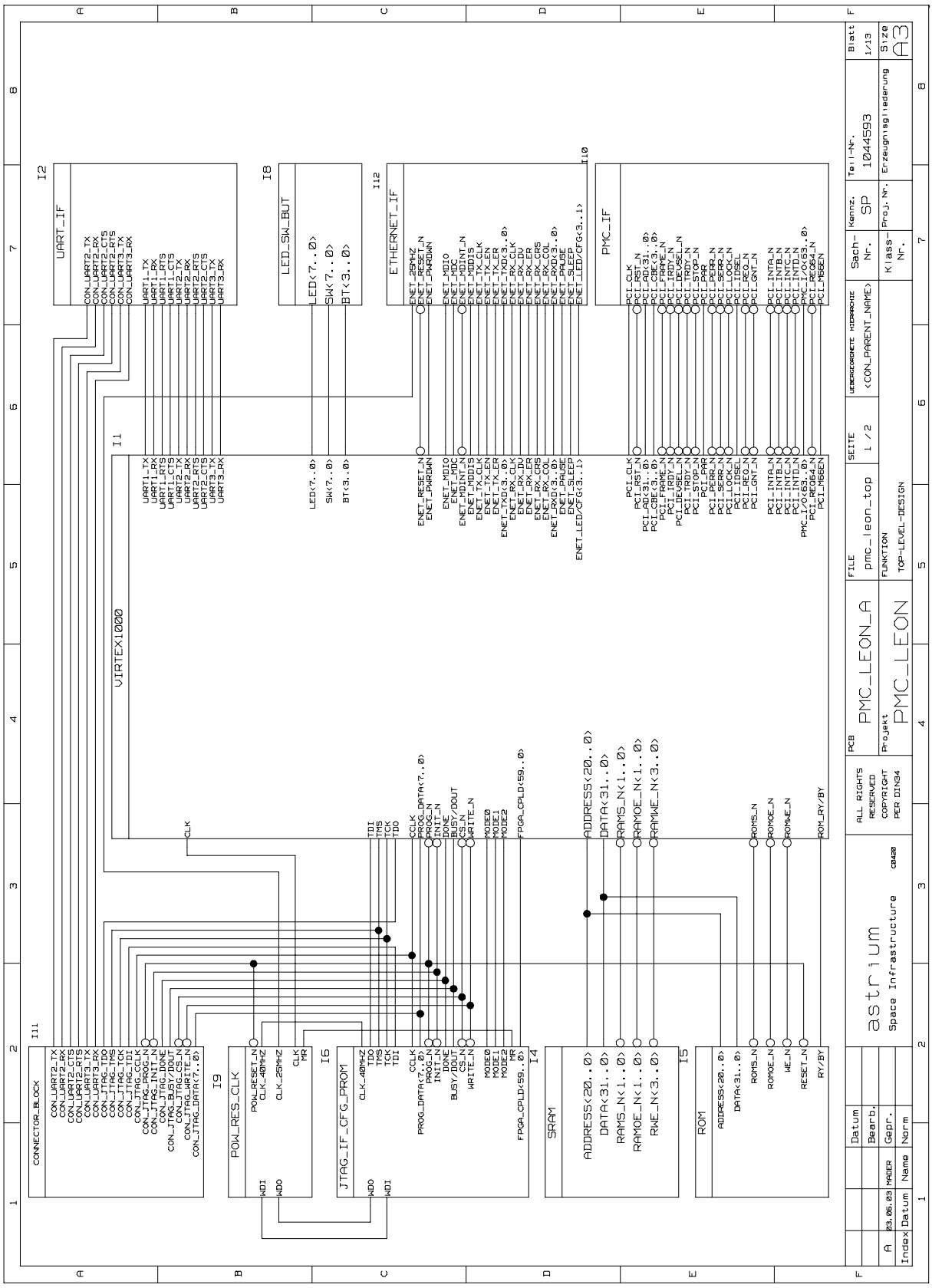
Application Notes

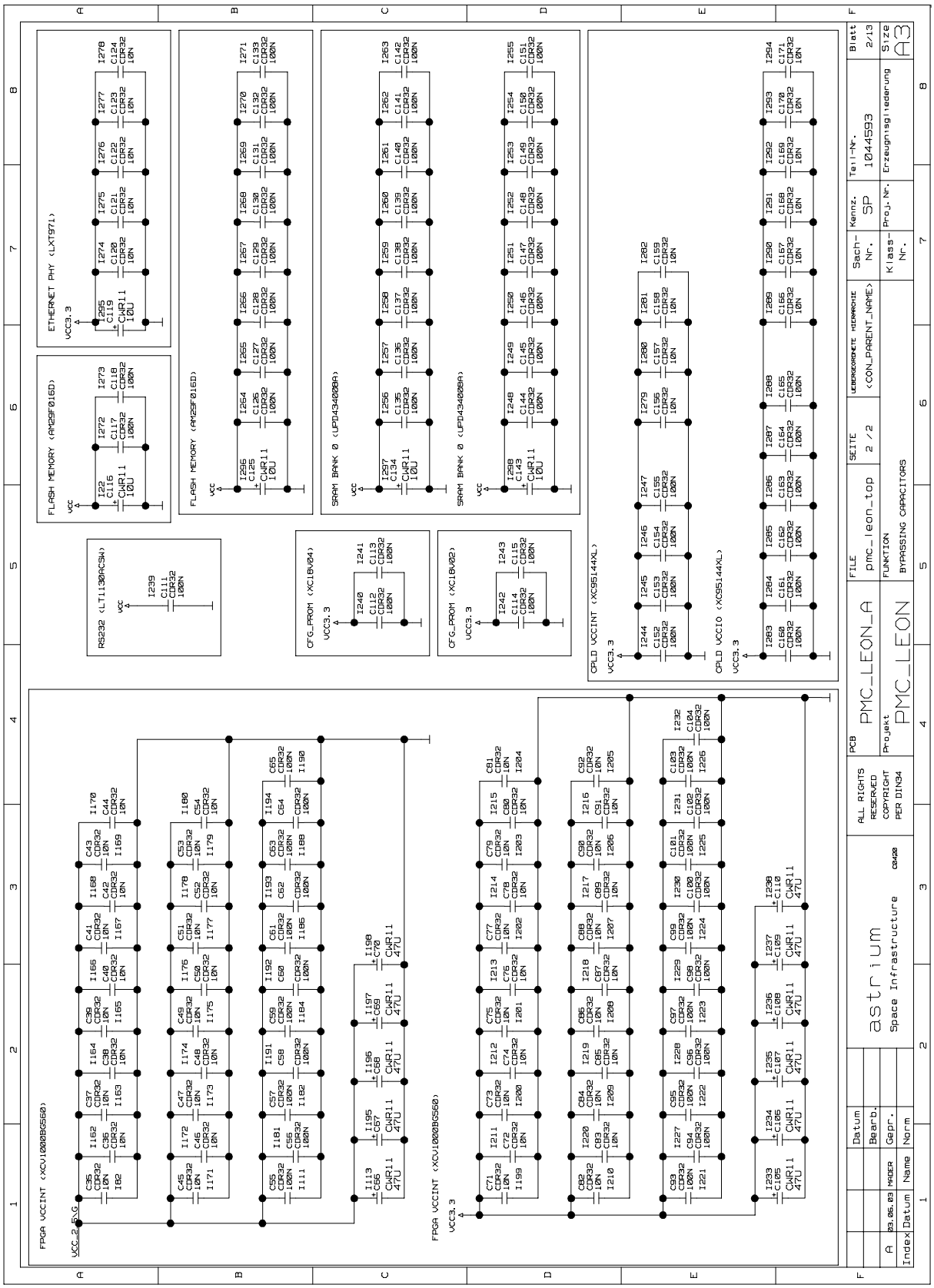
[A1] Xilinx, Inc.(XAPP079): Configuring Xilinx FPGAs Using an XC9500 CPLD and Parallel PROM [W1]
[A2] Xilinx, Inc.(XAPP137): Configuring Virtex FPGAs from Parallel EPROMs with a CPLD [W1]
[A3] Xilinx, Inc.(XAPP138): Virtex FPGA Series Configuration and Readback [W1]
[A4] Xilinx, Inc.(XAPP139): Configuration and Readback of Virtex FPGAs Using (JTAG) Boundary Scan [W1]
[A5] Xilinx, Inc.(XAPP158): Powering Xilinx FPGAs [W1]
[A6] Xilinx, Inc.(XAPP623): Power Distribution System (PDS) Design: Using Bypass/Decoupling Capacitors [W1]
[A7] Xilinx, Inc.(XAPP112): Designing With XC9500XL CPLDs [W1]
[A8] Intel Corporation Inc.: LXT971A/972A 3.3V PHY Transceivers Design and Layout Guide [W4]

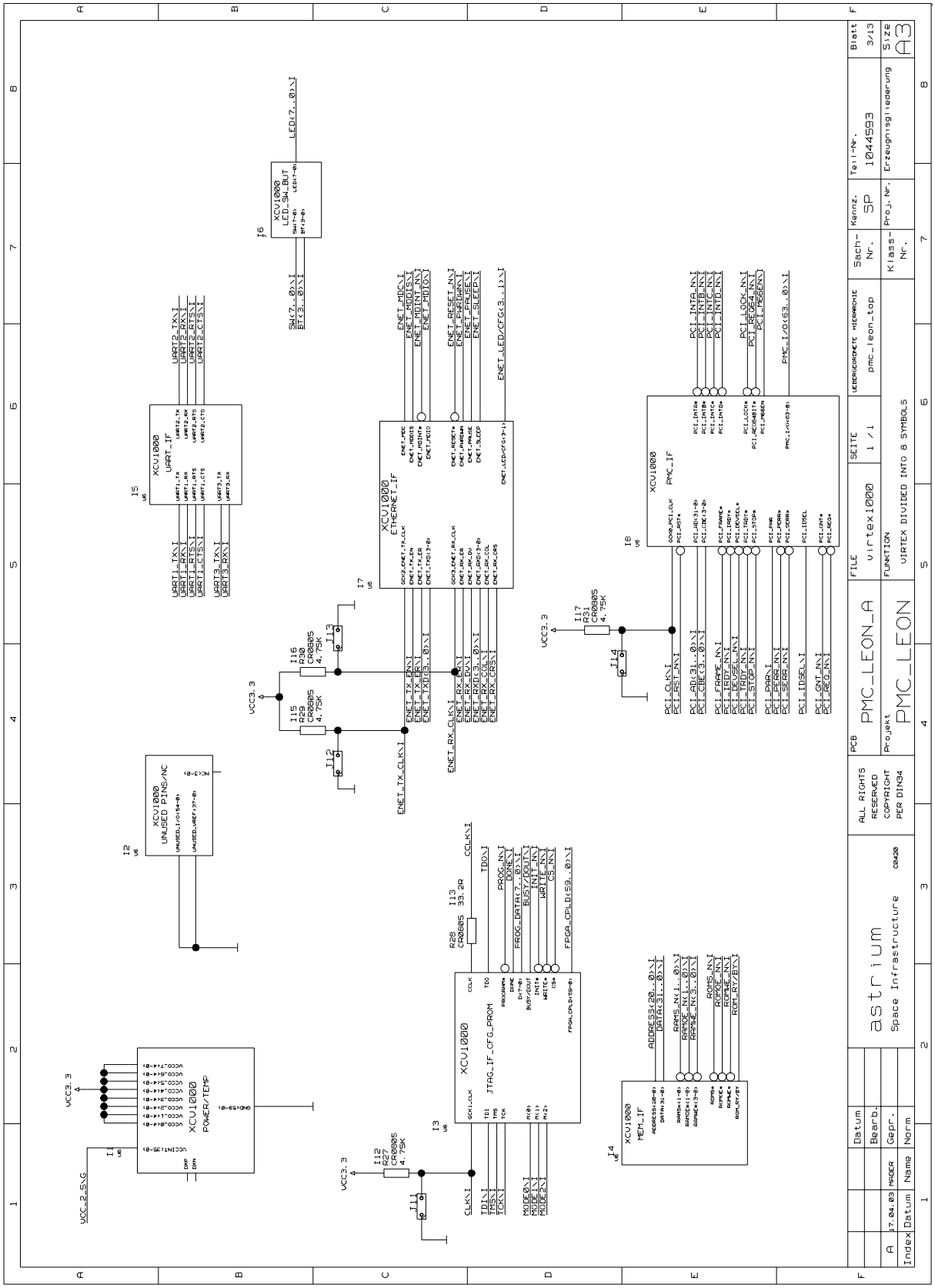
Internetadressen

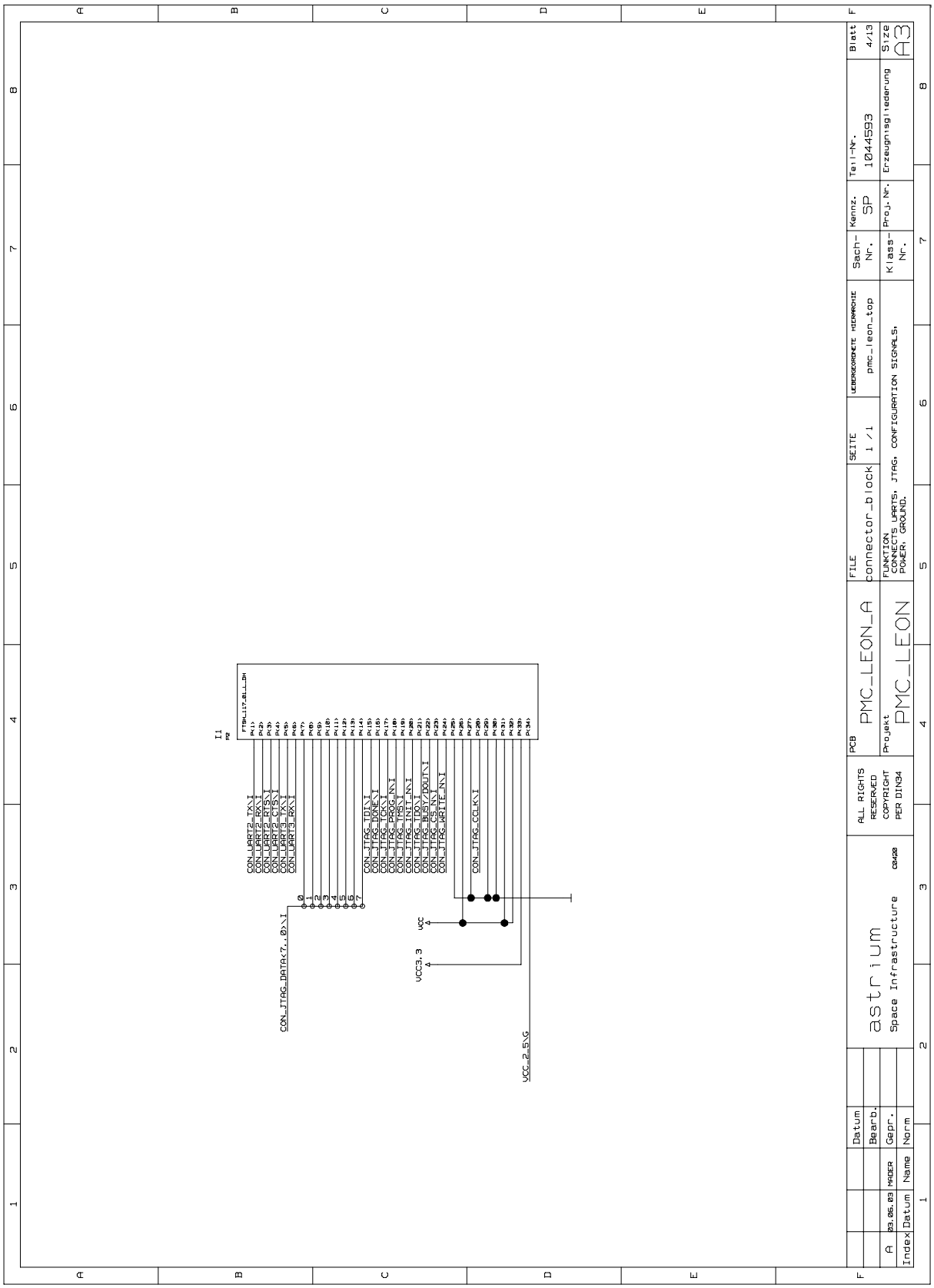
[W1] www.xilinx.com
[W2] www.pcisig.com
[W3] www.ieee.org
[W4] www.intel.com
[W5] www.gaisler.com
[W6] www.sparc.org
[W7] www.arm.com
[W8] www.kingbright-led.com
[W9] www.jauch.de
[W10] www.linear-tech.com
[W11] www.maxim-ic.com
[W12] www.amd.com
[W13] www.necel.com
[W14] www.tundra.com

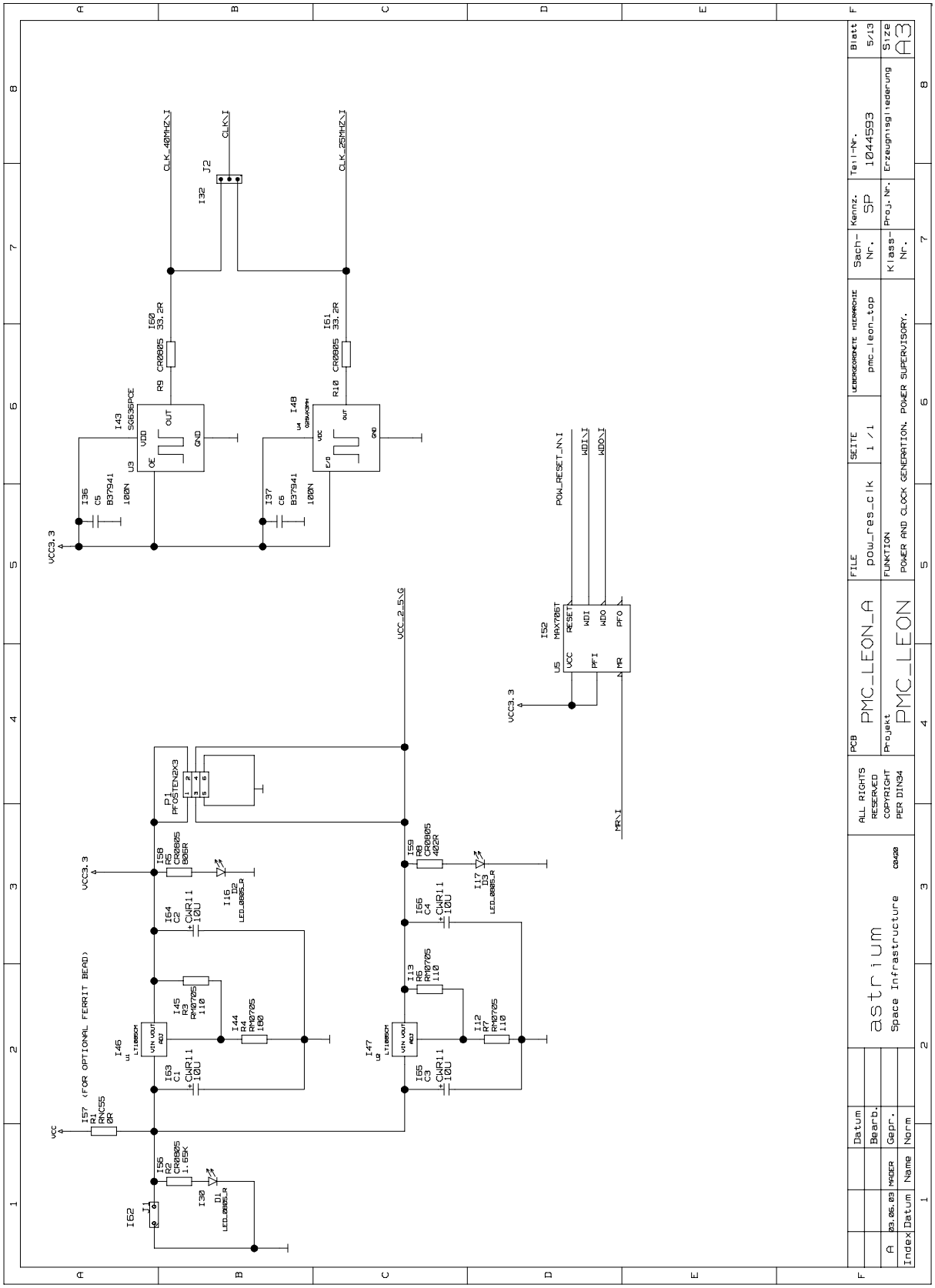
A-2 Schematics

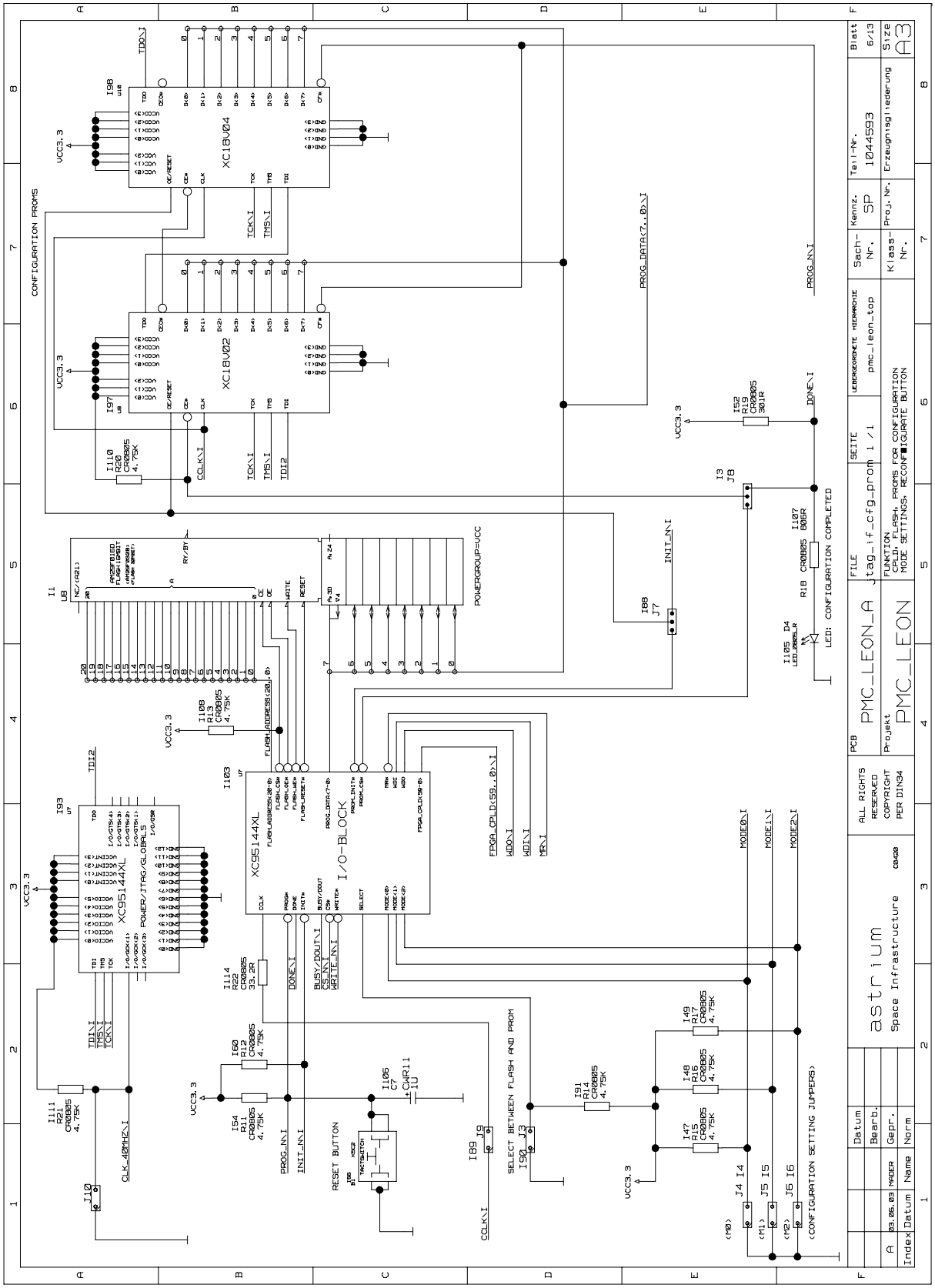


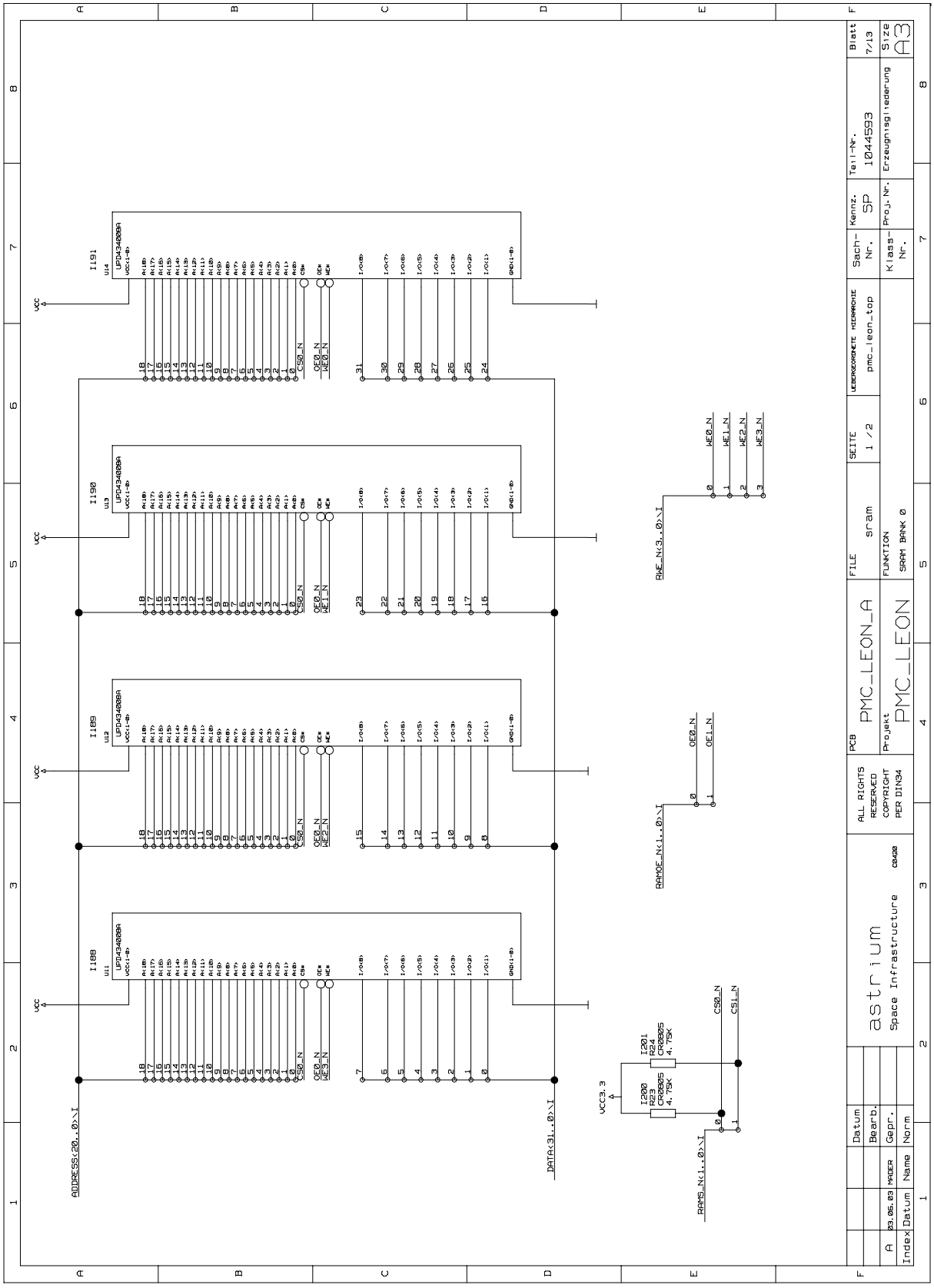

FPGA VCCINT (XCV1000BG560)
VCC_2.5_G
FPGA VCCINT (XCV1000BG560)
VCC3.3
FLASH MEMORY (AM29F016D)
ETHERNET PHY (LXT971)
FLASH MEMORY (AM29F016D)
SRAM BANK 0 (UPD434008A)
SRAM BANK 0 (UPD434008A)
RS232 (LT1130ACSW)
CFG_PROM (XC18V04)
CFG_PROM (XC18V02)
CPLD VCCINT (XC95144XL)
CPLD VCCIO (XC95144XL)
PCB PMC_LEON_A
FILE pmc_leon_top
SEITE 2 /2
FUNKTION BYPASSING CAPACITORS
Projekt PMC_LEON
astrium
Space Infrastructure
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
Sach-Nr. SP
Kennz. SP
Tail-Nr. 1044593
Blatt 2/13
Size A3
Erzeugnisgliederung

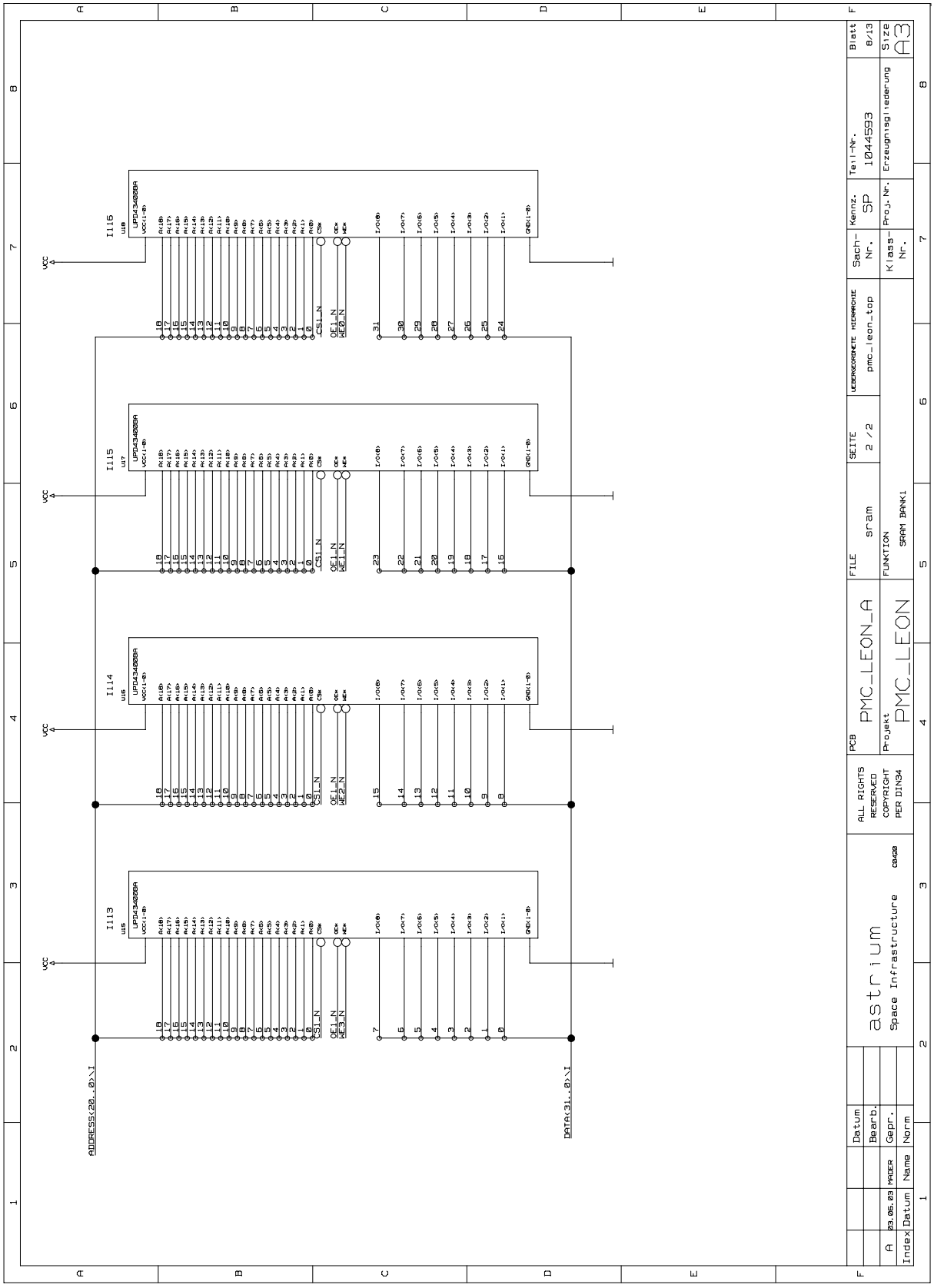

PMC_LEON_A
PMC_LEON
virtex1000
pmc_leon_top
VIRTEX DIVIDED INTO 8 SYMBOLS
astrium
Space Infrastructure
ALL RIGHTS RESERVED COPYRIGHT PER DIN34
XCV1000 POWER/TEMP
XCV1000 UNUSED PINS/NC
XCV1000 UART_IF
XCV1000 LED_SW_BUT
XCV1000 ETHERNET_IF
XCV1000 PMC_IF
XCV1000 JTAG_IF_CFG_PROM
XCV1000 MEM_IF
VCC3.3
CLK\
TDI\
TMS\
TCK\
MODE0\
MODE1\
MODE2\

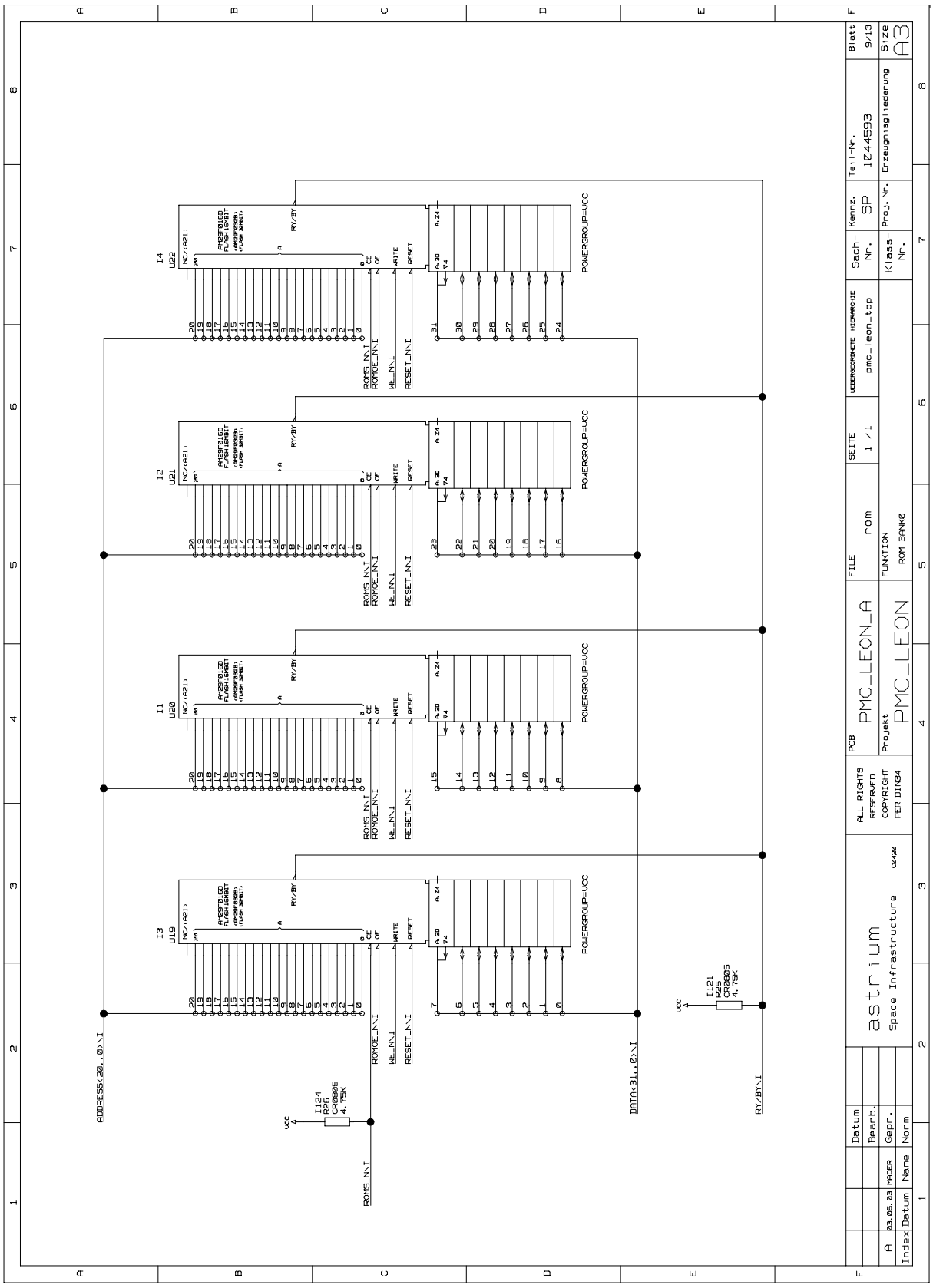

I1
FTSH-117-01-L-DH
CON_UART2_TX\.I
CON_UART2_RX\.I
CON_UART2_RTS\.I
CON_UART2_CTS\.I
CON_UART3_TX\.I
CON_UART3_RX\.I
CON_JTAG_TDI\.I
CON_JTAG_TDO\.I
CON_JTAG_TCK\.I
CON_JTAG_PROG_N\.I
CON_JTAG_TMS\.I
CON_JTAG_INIT_N\.I
CON_JTAG_TDO\.I
CON_JTAG_BUSY/DOUT\.I
CON_JTAG_CS_N\.I
CON_JTAG_WRITE_N\.I
CON_JTAG_CCLK\.I
CON_JTAG_DATA\(7..0\)\.I
VCC
VCC3.3
VCC_2.5\.G
PCB PMC_LEON_A
Projekt PMC_LEON
FILE connector_block
SEITE 1 / 1
FUNKTION
CONNECTS UARTS, JTAG, CONFIGURATION SIGNALS,
POWER, GROUND.
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
astrium
Space Infrastructure
Datum
Bearb.
Gepr.
Norm
MAIER
Name
23.05.03
Datum
Index
Sach-Nr.
Kennz. SP
Proj. Nr.
Klass-Nr.
Teil-Nr. 1044593
Erzeugnisgliederung
Blatt 4/13
Size A3

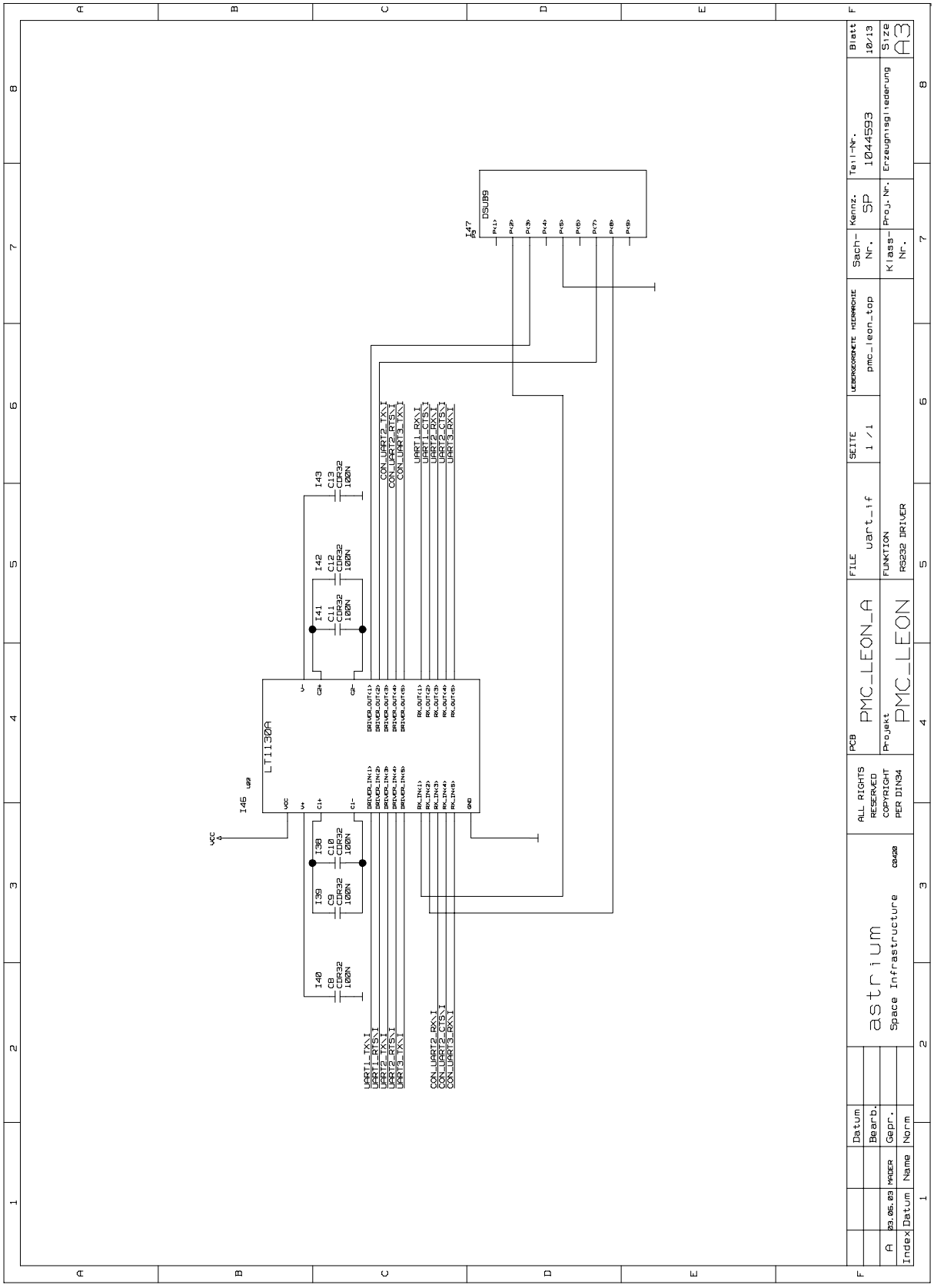

PCB PMC_LEON_A
FILE pow_res_clk
SEITE 1 / 1
UEBERGEORDNETE HIERARCHIE pmc_leon_top
Projekt PMC_LEON
FUNKTION POWER AND CLOCK GENERATION. POWER SUPERVISORY.
Teil-Nr. 1044593
Blatt 5/13
Size A3
astrium
Space Infrastructure
ALL RIGHTS RESERVED COPYRIGHT PER DIN34
CLK_40MHZ_I
CLK_I
CLK_25MHZ_I
POW_RESET_N_I
WDI_I
WDO_I
MR_I
VCC3.3
VCC_2_5_G
VCC
P1 PFOSTEN2X3
I52 MAX706T
I43 SG636PCE
I46 LT1880CN
I47 LT1880CN
I48 ORA2405H
I62 J1
I32 J2
I30 D1
I16 D2
I17 D3
I57 R1 RNC55 0R
I55 R2 CR0805 1.65K
I58 R5 CR0805 806R
I59 R6 CR0805 402R
I45 R3 RM0705 110
I44 R4 RM0705 150
I13 R6 RM0705 110
I12 R7 RM0705 110
I60 R9 CR0805 33.2R
I61 R10 CR0805 33.2R
I35 C5 B37941 100N
I37 C6 B37941 100N
I63 C1 CWR11 10U
I64 C2 CWR11 10U
I65 C3 CWR11 10U
I66 C4 CWR11 10U

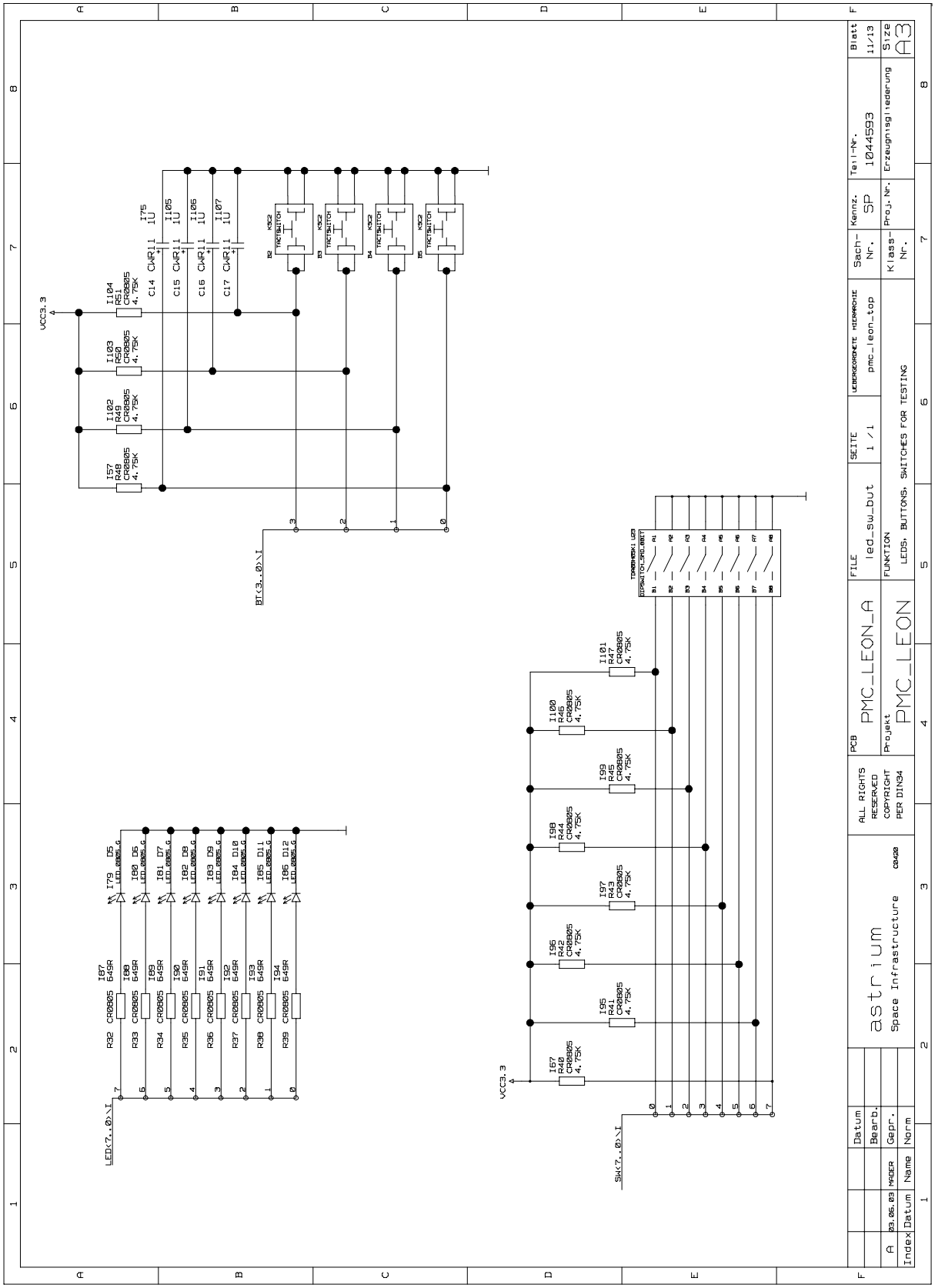

CONFIGURATION PROMS
XC18V04
XC18V02
XC95144XL
I/O-BLOCK
POWER/JTAG/GLOBALS
PROG_DATA<7..0>\I
PROG_N\I
DONE\I
INIT_N\I
TDI2
TCK\I
TMS\I
TDI\I
CCLK\I
BUSY/DOUT\I
CS_N\I
WRITE_N\I
FLASH_ADDRESS<20..0>
POWERGROUP=VCC
FPGA_CPLD<59..0>\I
KDO\I
KDI\I
MR\I
MODE0\I
MODE1\I
MODE2\I
CLK_40MHZ\I
RESET BUTTON
SELECT BETWEEN FLASH AND PROM
LED: CONFIGURATION COMPLETED
(CONFIGURATION SETTING JUMPERS)
PCB PMC_LEON_A
Projekt PMC_LEON
FILE jtag_if_cfg_prom
FUNKTION CPLD: FLASH: PROMS FOR CONFIGURATION MODE SETTINGS, RECONFIGURATE BUTTON
astrium
Space Infrastructure
ALL RIGHTS RESERVED COPYRIGHT PER DIN34
Blatt 6/13
Size A3
Tail-Nr. 1044593
Erzeugnisgliederung
Sach-Nr.
Kennz. SP
Proj. Nr.
Klass-Nr.
Datum
Bearb.
Gepr.
Norm
Index Datum Name
A 23.05.03 MADER

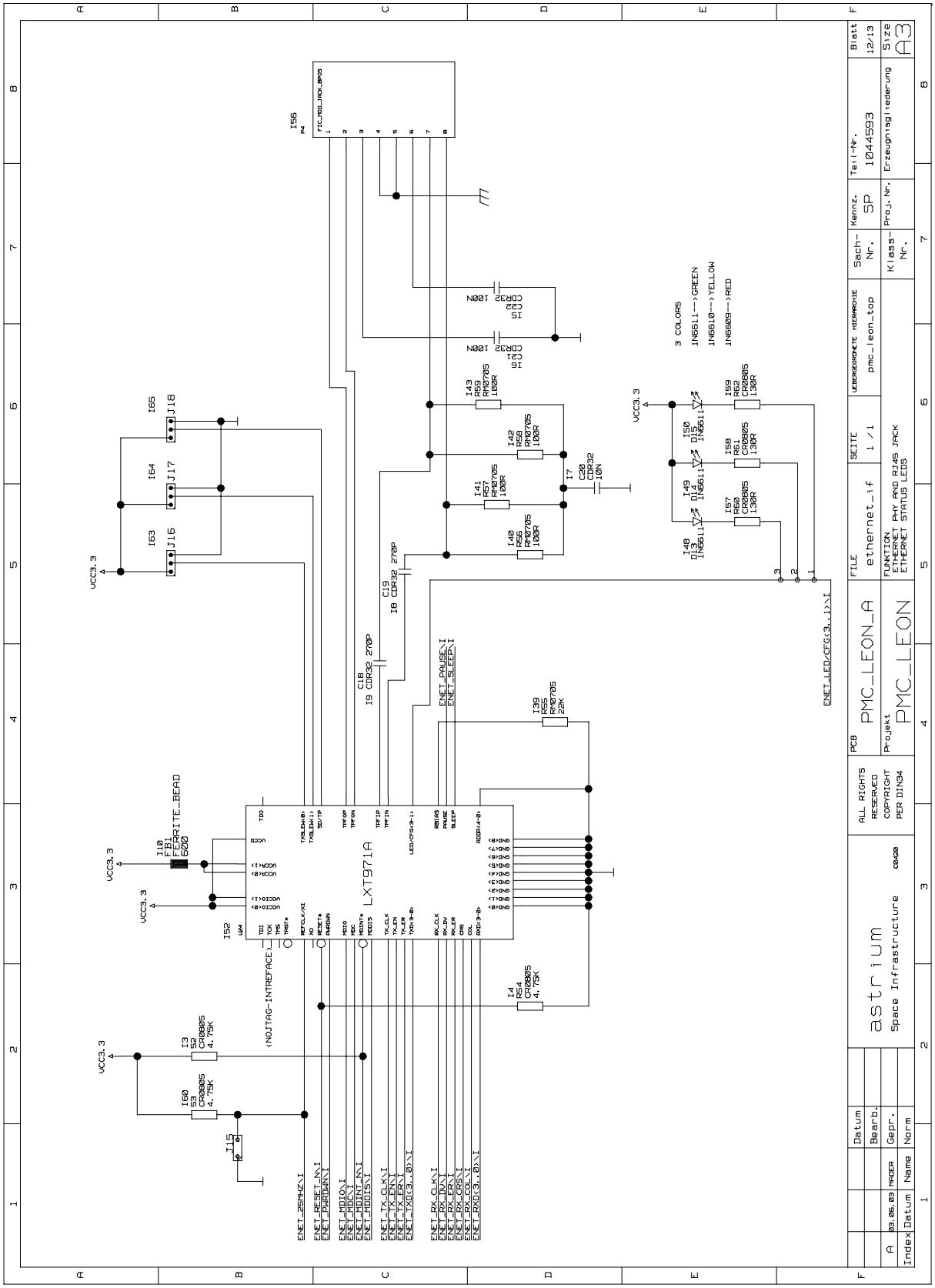

PMC_LEON_A
PMC_LEON
sram
SRAM BANK 0
pmc_leon_top
astrium
Space Infrastructure
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
ADDRESS<20..0>
DATA<31..0>
RAMS_N<1..0>
RAMOE_N<1..0>
RAE_N<3..0>
CS0_N
CS1_N
OE0_N
OE1_N
WE0_N
WE1_N
WE2_N
WE3_N
VCC3.3
R200 R201 R204 CR0805 4.75K

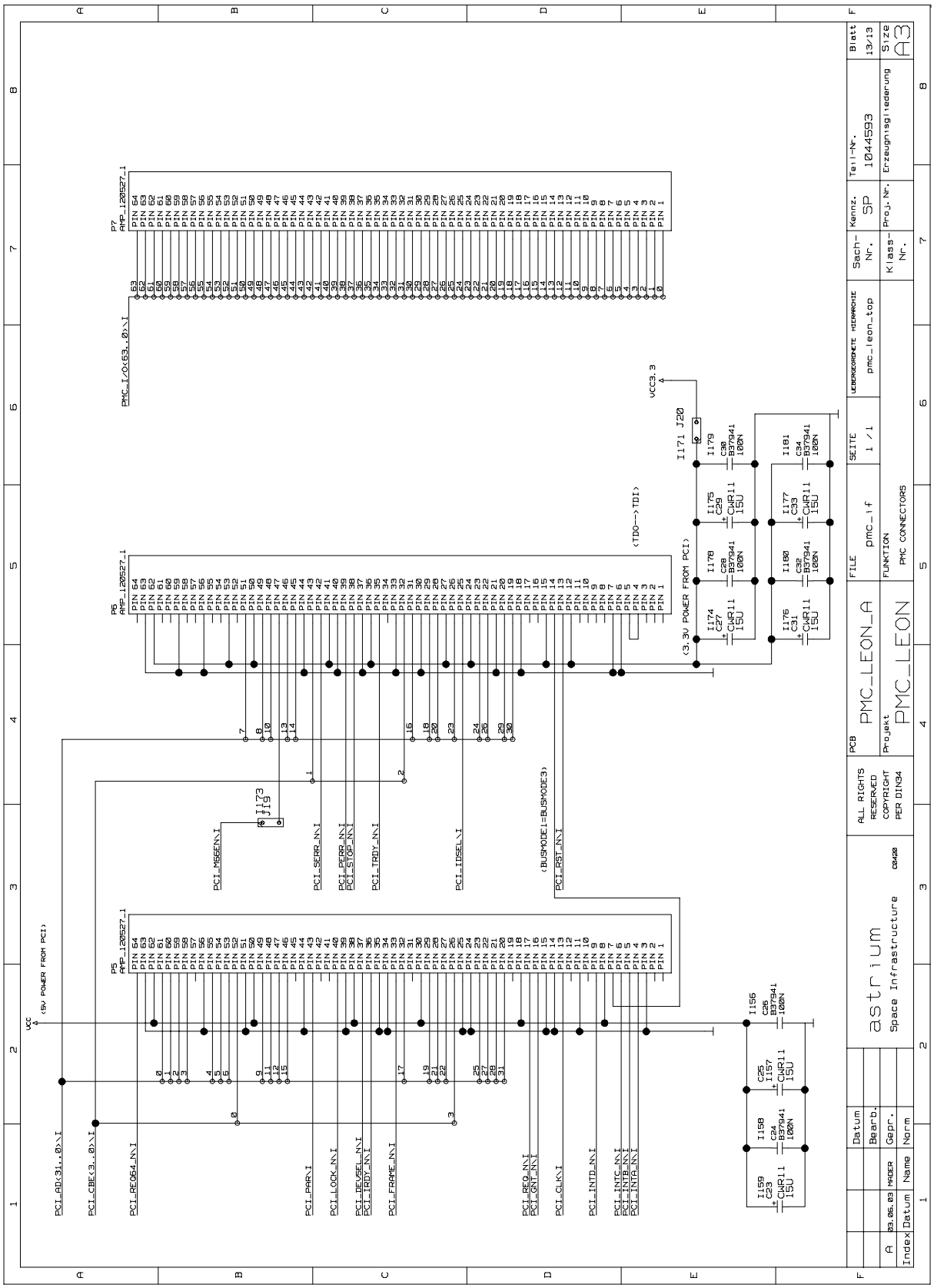

I116
I115
I114
I113
UPD434008A
ADDRESS<20..0>\I
DATA<31..0>\I
CS1_N
OE1_N
WE1_N
WE2_N
VCC
astrium
Space Infrastructure
PCB PMC_LEON_A
Projekt PMC_LEON
FILE sram
SEITE 2 /2
FUNKTION SRAM BANK1
pmc_leon_top
Blatt 8/13
Size A3
Tail-Nr. 1044593
Sach-Nr. SP
Kennz. SP
Index Datum Name
A 23.05.03 MADER
Datum
Bearb.
Gepr.
Norm
ALL RIGHTS RESERVED COPYRIGHT PER DIN34


PMC_LEON_A
PMC_LEON
rom
pmc_leon_top
ROM BANK0
astrium
Space Infrastructure
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
ADDRESS<20..0>
DATA<31..0>
RY/BY
ROMS_N
ROMOE_N
WE_N
RESET_N
POWERGROUP=VCC
AM29F016D FLASH 16MBIT


LT1130A
DSUB9
PMC_LEON_A
PMC_LEON
uart_if
RS232 DRIVER
pmc_leon_top
astrium
Space Infrastructure
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
Blatt 10/13
Size A3


PMC_LEON_A
PMC_LEON
led_sw_but
pmc_leon_top
LEDS, BUTTONS, SWITCHES FOR TESTING
astrium
Space Infrastructure
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
VCC3.3
BT<3..0>
LED<7..0>
SW<7..0>


PCB PMC_LEON_A
Projekt PMC_LEON
FILE ethernet.if
SEITE 1 / 1
FUNKTION ETHERNET PHY AND RJ45 JACK
ETHERNET STATUS LEDS
astrium
Space Infrastructure
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
Blatt 12/13
Size A3
Tail-Nr. 1044593
Erzeugnisgliederung
Kennz. SP
Sach-Nr.
Klass-Nr.
Proj.Nr.
LXT971A
3 COLORS
IN6611-->GREEN
IN6610-->YELLOW
IN6609-->RED
(NOJTAG-INTREFACE)
FERRITE_BEAD
ENET_25MHZ_I
ENET_RESET_NN_I
ENET_PWRDWN_I
ENET_MDIO_I
ENET_MDC_I
ENET_MDINT_NN_I
ENET_MDDIS_I
ENET_TX_CLK_I
ENET_TX_EN_I
ENET_TX_ER_I
ENET_TXD[3..0]_I
ENET_RX_CLK_I
ENET_RX_DV_I
ENET_RX_ER_I
ENET_RX_CRS_I
ENET_RX_COL_I
ENET_RXD[3..0]_I
ENET_PAUSE_I
ENET_SLEEP_I
ENET_LED/CFG<3..1>_I


PMC_LEON_A
PMC_LEON
pmc_if
pmc_leon_top
PMC CONNECTORS
astrium
Space Infrastructure
ALL RIGHTS RESERVED
COPYRIGHT PER DIN34
PCI_AD<31..0>_N_I
PCI_CBE<3..0>_N_I
PCI_REQ64_N_I
PCI_PAR_I
PCI_LOCK_N_I
PCI_DEVSEL_N_I
PCI_IRDY_N_I
PCI_FRAME_N_I
PCI_REQ_N_I
PCI_GNT_N_I
PCI_CLK_I
PCI_INTD_N_I
PCI_INTC_N_I
PCI_INTB_N_I
PCI_INTA_N_I
PCI_M66EN_I
PCI_SERR_N_I
PCI_PERR_N_I
PCI_STOP_N_I
PCI_TRDY_N_I
PCI_IDSEL_I
PCI_RST_N_I
PMC_I/O<63..0>_I
(5V POWER FROM PCI)
(3.3V POWER FROM PCI)
(TDO->TDI)
(BUSMODE1=BUSMODE3)
VCC
VCC3.3

A-3 Quellcodes

In diesem Abschnitt sind die Quellcodes der Testprogramme sowie des Boot Loaders zu finden.

Boot Loader

```
/*****************************************************************/
/* Compile with LECCS:
   sparc-rtems-gcc -nostdlib -nostdinc -O2 -Ttext=0 bprom.c -o bprom
*/
/*****************************************************************/

/* LEON register layout */
struct lregs {
      volatile unsigned int memcfg1;           /* 0x00 */
      volatile unsigned int memcfg2;
      volatile unsigned int ectrl;
      volatile unsigned int failaddr;
      volatile unsigned int memstatus;         /* 0x10 */
      volatile unsigned int cachectrl;
      volatile unsigned int powerdown;
      volatile unsigned int writeprot1;
      volatile unsigned int writeprot2;  /* 0x20 */
      volatile unsigned int leonconf;
      volatile unsigned int dummy2;
      volatile unsigned int dummy3;
      volatile unsigned int dummy4;      /* 0x30 */
      volatile unsigned int dummy5;
      volatile unsigned int dummy6;
      volatile unsigned int dummy7;
      volatile unsigned int timercnt1;         /* 0x40 */
      volatile unsigned int timerload1;
      volatile unsigned int timerctrl1;
      volatile unsigned int wdog;
      volatile unsigned int timercnt2;
      volatile unsigned int timerload2;
      volatile unsigned int timerctrl2;
      volatile unsigned int dummy8;
      volatile unsigned int scalercnt;
      volatile unsigned int scalerload;
      volatile unsigned int dummy9;
      volatile unsigned int dummy10;
      volatile unsigned int uartdata1;
      volatile unsigned int uartstatus1;
      volatile unsigned int uartctrl1;
      volatile unsigned int uartscaler1;
      volatile unsigned int uartdata2;
      volatile unsigned int uartstatus2;
      volatile unsigned int uartctrl2;
      volatile unsigned int uartscaler2;
      volatile unsigned int irqmask;
      volatile unsigned int irqpend;
      volatile unsigned int irqforce;
      volatile unsigned int irqclear;
      volatile unsigned int piodata;
      volatile unsigned int piodir;
      volatile unsigned int pioirq;
};
```

```
/* boot assembly code, setup %wim, %psr and stack */
/* NOTE: no window handling, maximum function call-depth is 7 !! */

asm("

  .global _start


_start:
        wr      %g0, 0xc0, %psr           ! disable traps, enable 0xe0
      wr      %g0, 2, %wim
      flush
      set   0x10000, %g1            ! no enable cache, enable:0x1000f
      set   0x80000000, %o0         ! register base
      sub   %o0, 16, %g4            ! top of ram
      st    %g1, [%o0 + 0x14] ! no enable cache
      ld    [%o0 + 0x24], %o1 ! probe for FPU
      and   %o1, 0x10, %o1
      sll   %o1, 8, %o1
        wr      %o1, 0xc0, %psr             !no trap enable  ,enable 0xe0

      set   0xaa00, %o1       ! enable both uarts
      st    %o1, [%o0 + 0xA4]
      st    %g0, [%o0 + 0x74]
      st    %g0, [%o0 + 0x84]
      set   3, %o1
      st    %o1, [%o0 + 0x78]
      st    %o1, [%o0 + 0x88]



! new
        set     0x333, %o1              ! set MCFG1
        st      %o1, [%o0 + 0x0]

        set     0x182000, %o1           ! set MCFG3
        st      %o1, [%o0 + 0x8]
        set     0xe2386020, %o1         ! set MCFG2
        st      %o1, [%o0 + 0x4]
        set     0x0, %o1                ! reset AHB status register
        st      %o1, [%o0 + 0x10]
        set     0x50, %o1               ! set uart1&2 scaler
        st      %o1, [%o0 + 0x7c]
        st      %o1, [%o0 + 0x8c]

        set     0x43fffff0, %sp         ! set stack pointer

        or      %g0, bmsg1, %o0         !print bmsg1
        call    puts
        nop
      call  puts
      or    %g0, bmsg4, %o0
        nop
        call    boot
        nop
        nop
");

const char bmsg1[1][9] = {"LEON-1: "};
//const char bmsg2[1][10]={"completed"};
const char bmsg4[1][6] = {"\n\n\r> "};
```

```
puts(unsigned char *s) {
      struct lregs *regs = (struct lregs *) 0x80000000;

      while(*s) {
            while (!(regs->uartstatus1 & 0x4));
            regs->uartdata1 = *s++;
      }
}

static inline gets(unsigned char *s) {
      struct lregs *regs = (struct lregs *) 0x80000000;
      do {
            while (!(regs->uartstatus1 & 0x1));
            *s = regs->uartdata1;
            if ((*s == '\r') || (*s == '\n')) {
                  break;
            }
            s++;
      } while (1);
}

static
unsigned int h2i(int len, unsigned char *h)
{
      unsigned int tmp = 0;
      unsigned int i;
      unsigned char c;

      for (i=0; i<len; i++) {
          c = h[i];
          if (c >= 'A') c = c - 'A' + 10; else c -= '0';
          tmp = (tmp << 4) | c;
      }
      return(tmp);
}

boot() {
      unsigned char r[256];
      int i, size, addr;
      char *data;
        void (*prog) ();
      double dummy;
      unsigned char *bmsg2="transfer completed \n\n\r";
      unsigned char *anfang="end initialization \n\n\r";
      unsigned char *ende="program completed  \n\n\r";

      puts(anfang);
      do {
          gets(r);
          if (r[0] == 'S') {
            addr = h2i(8, &r[4]);
            data = (unsigned char *) &r[12];
            if (r[1] == '3') {
                size = (h2i(2, &r[2])*2 - 10) >> 1;
                for (i=0;i<size;i++) ((char *)addr)[i] = h2i(2,&data[2*i]);
            } else if (r[1] == '7') {
              //new
              puts(bmsg2);
                prog = (void *) addr;
                prog();
                puts(ende);
            }
          }
```

```
	} while (1);
}
```

UART 1-Test:

```
/* new hello word program */

/* LEON register layout */
struct lregs {
	volatile unsigned int memcfg1;		/* 0x00 */
	volatile unsigned int memcfg2;
	volatile unsigned int ectrl;
	volatile unsigned int failaddr;
	volatile unsigned int memstatus;	/* 0x10 */
	volatile unsigned int cachectrl;
	volatile unsigned int powerdown;
	volatile unsigned int writeprot1;
	volatile unsigned int writeprot2;	/* 0x20 */
	volatile unsigned int leonconf;
	volatile unsigned int dummy2;
	volatile unsigned int dummy3;
	volatile unsigned int dummy4;		/* 0x30 */
	volatile unsigned int dummy5;
	volatile unsigned int dummy6;
	volatile unsigned int dummy7;
	volatile unsigned int timercnt1;	/* 0x40 */
	volatile unsigned int timerload1;
	volatile unsigned int timerctrl1;
	volatile unsigned int wdog;
	volatile unsigned int timercnt2;
	volatile unsigned int timerload2;
	volatile unsigned int timerctrl2;
	volatile unsigned int dummy8;
	volatile unsigned int scalercnt;
	volatile unsigned int scalerload;
	volatile unsigned int dummy9;
	volatile unsigned int dummy10;
	volatile unsigned int uartdata1;
	volatile unsigned int uartstatus1;
	volatile unsigned int uartctrl1;
	volatile unsigned int uartscaler1;
	volatile unsigned int uartdata2;
	volatile unsigned int uartstatus2;
	volatile unsigned int uartctrl2;
	volatile unsigned int uartscaler2;
	volatile unsigned int irqmask;
	volatile unsigned int irqpend;
	volatile unsigned int irqforce;
	volatile unsigned int irqclear;
	volatile unsigned int piodata;
	volatile unsigned int piodir;
	volatile unsigned int pioirq;
};

puts(unsigned char *s) {
	struct lregs *regs = (struct lregs *) 0x80000000;

	while(*s) {
		while (!(regs->uartstatus1 & 0x4));
		regs->uartdata1 = *s++;
	}
}
```

```
void _start()
{unsigned char *text="Hello World \n\n\r";
 int i;

 for (i=0;i<5;i++)
   {puts(text);};
};
```

io_leon.h:

Diese Header-Datei wird benutzt, um ständige Wiederholungen der Ein- und Ausgabe-Routinen der UARTs zu vermeiden. Sie wird in den Testprogrammen eingebunden (mit Ausnahme von „Hello world“).

```
/* LEON register layout */
struct lregs {
      volatile unsigned int memcfg1;           /* 0x00 */
      volatile unsigned int memcfg2;
      volatile unsigned int ectrl;
      volatile unsigned int failaddr;
      volatile unsigned int memstatus;         /* 0x10 */
      volatile unsigned int cachectrl;
      volatile unsigned int powerdown;
      volatile unsigned int writeprot1;
      volatile unsigned int writeprot2;  /* 0x20 */
      volatile unsigned int leonconf;
      volatile unsigned int dummy2;
      volatile unsigned int dummy3;
      volatile unsigned int dummy4;      /* 0x30 */
      volatile unsigned int dummy5;
      volatile unsigned int dummy6;
      volatile unsigned int dummy7;
      volatile unsigned int timercnt1;         /* 0x40 */
      volatile unsigned int timerload1;
      volatile unsigned int timerctrl1;
      volatile unsigned int wdog;
      volatile unsigned int timercnt2;
      volatile unsigned int timerload2;
      volatile unsigned int timerctrl2;
      volatile unsigned int dummy8;
      volatile unsigned int scalercnt;
      volatile unsigned int scalerload;
      volatile unsigned int dummy9;
      volatile unsigned int dummy10;
      volatile unsigned int uartdata1;
      volatile unsigned int uartstatus1;
      volatile unsigned int uartctrl1;
      volatile unsigned int uartscaler1;
      volatile unsigned int uartdata2;
      volatile unsigned int uartstatus2;
      volatile unsigned int uartctrl2;
      volatile unsigned int uartscaler2;
      volatile unsigned int irqmask;
      volatile unsigned int irqpend;
      volatile unsigned int irqforce;
      volatile unsigned int irqclear;
      volatile unsigned int piodata;
      volatile unsigned int piodir;
      volatile unsigned int pioirq;
};
```

```
puts(unsigned char *s) {
      struct lregs *regs = (struct lregs *) 0x80000000;

      while(*s) {
            while (!(regs->uartstatus1 & 0x4));
            regs->uartdata1 = *s++;
      }
}

static inline gets(unsigned char *s) {
      struct lregs *regs = (struct lregs *) 0x80000000;
      do {
            while (!(regs->uartstatus1 & 0x1));
            *s = regs->uartdata1;
            if ((*s == '\r') || (*s == '\n')) {
                  break;
            }
            s++;
      } while (1);
}

putsb(unsigned char *s) {
      struct lregs *regs = (struct lregs *) 0x80000000;

      while(*s) {
            while (!(regs->uartstatus2 & 0x4));
            regs->uartdata2 = *s++;
      }
}

static inline getsb(unsigned char *s) {
      struct lregs *regs = (struct lregs *) 0x80000000;
      do {    while (!(regs->uartstatus2 & 0x1));
            *s = regs->uartdata2;
            if ((*s == '\r') || (*s == '\n')) {
                  break;
            }
            s++;
      } while (1);
}
```

UART 2-Test:

```
/*********************************************************************/
//test of UART 2
/*********************************************************************/
#include "io_leon.h"

void _start()
{
 unsigned char *bmsg1="UART2 listening. \n\n\r";
 unsigned char *bmsg2="You sent: \n\n\r";
 unsigned char *bmsg3="\n\n\r";
 unsigned char r[256];
 unsigned int i;


 for(i=0; i<256; i++) r[i]=0;//clear space for reading

 puts(bmsg1);
 putsb(bmsg1);
```

```
 while(r[0]<0x20)getsb(r);//wait for lines beginning without control signs
 puts(bmsg2);
 putsb(bmsg2);
 puts(r);
 putsb(r);
 puts(bmsg3);
 putsb(bmsg3);

};
```

PIO-Test:

```
/*****************************************************************/
// PIO test
/*****************************************************************/
#include "io_leon.h"

void _start()
{
 unsigned char *bmsg1="Program aborted by user \n\n\r";
 unsigned char *bmsg2="PIO(6) is HIGH \n\n\r";
 unsigned int i;
 unsigned int y;
 unsigned char pio_byte;
 unsigned char help;

 struct lregs *regs = (struct lregs *) 0x80000000;

 y = 0;

 i = (regs->piodir );//read pio direction register
 i = i & 0xFFFFFF0F; // pio (4..7) as input
 i = i | 0xC;        // pio (2..3) as output
 regs->piodir = i;   // set direction register

do
   {
     for (i=0; i<1000; i++) //wait for some time
       {;};
     help = 0;
     pio_byte = regs->piodata;//read Lowest Byte from pio data register

     if ((pio_byte & 0x20) > 1)//pio(5) sets pio(3)
       help = 0x08;
     if ((pio_byte & 0x10) > 1)//pio(4) sets pio(2)
       help = help | 0x04;
     regs->piodata = help;//assign output values

     if (((pio_byte & 0x40) > 1)&& y==0) // pio(6)produces output on uart1
//(one time per use of pio(6))
       {puts(bmsg2);
       y =1;};
     if ((pio_byte & 0x40) == 0) y=0;
     if ((pio_byte & 0x80)> 1)// pio(7) as program abort
       {
       puts(bmsg1);
       break;};
   }while(1);

}
```

FLASH-Test:

```
/*************************************************************/
// test of FLASH on AVNET-Board
/*************************************************************/

#include "io_leon.h"
#define flash_start 0x10000000;


/* for flash addresses : if programming in C,you'll have to
add the FLASH-Adresses to the Flash base address, like it is
described in the flash manual.
The C-Compiler multiplies them with 4 because the pointer
flash_begin is of size int.
The Memory Controller of the LEON translates them back to the
address you add here because the controller only "thinks" of flash
devices with 8 bit width.*/

void _wait(unsigned int time)
{ int help;
  struct lregs *regs = (struct lregs *) 0x80000000;
  help = regs->irqmask;
  regs->irqmask = (help & 0xFFFFFDFF);//mask timer2 irq
  regs->timerctrl2 =0x0;// disable timer2
  regs->irqclear = 0x00000200; //clear timre2 irq(could be pending)
  regs->scalerload = 0x1B; //about 1 us
  regs->timerload2 = time; // time specifies the wait time in us
  regs->timerctrl2 = 0x5; //load and enable timer2
  while(((regs->irqpend)&0x00000200)==0);
  regs->irqclear = 0x00000200;
}

void _erase_flash_sector(unsigned int SA)
{
  int *flash_begin =(int *)flash_start;

  /* erase sequence */
  *(flash_begin + 0x555)= 0x00aa00aa;
  *(flash_begin + 0x2aa)= 0x00550055;
  *(flash_begin + 0x555)= 0x00800080;
  *(flash_begin + 0x555)= 0x00aa00aa;
  *(flash_begin + 0x2aa)= 0x00550055;
  *(flash_begin +(SA <<= 15) )= 0x00300030;
  _wait(2000000); //wait 2 sec
}

void _erase_flash()
{
  int *flash_begin =(int *)flash_start;

  /* erase flash */
  *(flash_begin + 0x555)= 0x00aa00aa;
  *(flash_begin + 0x2aa)= 0x00550055;
  *(flash_begin + 0x555)= 0x00800080;
  *(flash_begin + 0x555)= 0x00aa00aa;
  *(flash_begin + 0x2aa)= 0x00550055;
  *(flash_begin + 0x555)= 0x00100010;
  _wait(180000000);//wait at least 3 min
}

void _reset_flash()
```

```
{
  int *flash_begin =(int *)flash_start;

  *flash_begin = 0x00f000f0;
 _wait(12);//wait at least 11 usec
}

void _program(unsigned int offset, int value)
{
 int *flash_begin =(int *)flash_start;

 /*program sequence*/
 *(flash_begin + 0x555)= 0x00aa00aa;
  *(flash_begin + 0x2aa)= 0x00550055;
  *(flash_begin + 0x555)= 0x00a000a0;
  *(flash_begin + (offset>>=2)) = value;
 _wait(12);//wait at least 11 usec
}

void _unlock_bypass()
{
  int *flash_begin =(int *)flash_start;

  /*unlock flash*/
  *(flash_begin + 0x555)= 0x00aa00aa;
  *(flash_begin + 0x2aa)= 0x00550055;
  *(flash_begin + 0x555)= 0x00200020;
  _wait(12);//wait at least 11 usec
}

void _ub_program(unsigned int offset, int value)
{
  int *flash_begin =(int *)flash_start;

  *flash_begin= 0x00a000a0;
  *(flash_begin + (offset>>=2)) = value;
   _wait(12);//wait at least 11 usec
}

void _ub_reset()
{
  int *flash_begin =(int *)flash_start;
  *flash_begin= 0x00900090;
  *flash_begin= 0x00000000;
  _wait(12);//wait at least 11 usec
}

void _start()
{ unsigned char r[256];
  int i, size, addr, end_addr, sectors, start_addr, start_addr_set;
  int length, div;
  char *data;
  struct lregs *regs = (struct lregs *) 0x80000000;
  unsigned int *value;
  unsigned int failure;
  int *flash_begin =(int *)flash_start;

  regs->memcfg1 = 0x00000a33; //set 3 cycle wait and write enable

  /*get data file*/

  puts("Send SREC-File \n\n\r");
  start_addr_set = 0;
```

```
  do {
          gets(r);
          if (r[0] == 'S') {
            addr = h2i(8, &r[4]);
            data = (unsigned char *) &r[12];
            if (r[1] == '3') {
            if(start_addr_set == 0)
              {start_addr = addr;
              start_addr_set =1;
              };
            size = (h2i(2, &r[2])*2 - 10) >> 1;
            end_addr =addr+ size-1;
            for (i=0;i<size;i++) ((char *)addr)[i] = h2i(2,&data[2*i]);
            } else if (r[1] == '7') {
            puts("File received\n\n\r");
            break;
            }
          }
      } while (1);

  /* calculate number of used sectors in flash
   0x20000 address space on AHB per sector*/
  size = end_addr - start_addr;
  sectors = (size / 0x20000);//sector 0 is always used
  puts("Starting erasing\n\n\r");
  _reset_flash();
  if(sectors > 100){_erase_flash();}
  else{
    for(i=0;i<=sectors;i++){_erase_flash_sector(i);};
  };

  /* copy data in flash */
  puts("Starting copying \n\n\r");
  size >>=2; //size in words

  _unlock_bypass();
  value=(unsigned int *)start_addr;
  for(i=0;i<=size;i++)
    {
      _ub_program((i*4),*(value+i));
    };
  _ub_reset();

  /*verify*/
  puts("Verifying\n\n\r");

  failure = 0;
  value=(unsigned int *)start_addr;
  for(i=0;i<=size;i++)
    {
      if(*(flash_begin+i)!=*(value+i))
      {failure++;};
      };

  puts("Verifying completed: ");
  if (failure == 0)puts("no errors\n\n\r");
  else puts(" errors\n\n\r");

}
```

PCI-Test:

```
/*************************************************************************
*****/
// test of PCI-Interface
/*************************************************************************
****/

#include "io_leon.h"
#define pci_addr 0xA0000000
#define bridge_cfg_addr 0x80000100

void _start()
{ unsigned char r[256];
  int i,k, size, addr, last_addr, first_addr, sectors, start_addr,
first_addr_set;
  char *data;
  struct lregs *regs = (struct lregs *) 0x80000000;
  unsigned int *value;
  unsigned int failure= 1;
  unsigned int *bridge_cfg =(int *)bridge_cfg_addr;
  unsigned int *pci = (int *)pci_addr;
  unsigned int help, id, mem;


  /*disable ahb-error-generation*/
  *(bridge_cfg+0x3f)=0x0;//disable error and retry generation in register
0xff

  /*look if bridge is enabled as master*/
  help = *(bridge_cfg+0x1);
  if ((help & 0x4)==0)
    {*(bridge_cfg+0x1)=help |0x4;};


  /*look for memory controller*/

  *(bridge_cfg+0x3e)=0x5; //set remap register to cfg-access register 0xf8

  for(i=0;i <16; i++)     //get ID of Memory controller
    {


      help = *(pci+(i<<9)+0x2);//read class code
      if((*(bridge_cfg+0x3f)&0x0000000c)>0)
      {*(bridge_cfg+0x3f)=0x0;
      }
      if ((help & 0xFFFFFF00)==0x05800000)
      {id = (i<<9);
      puts("Memory Controller found\n\n\r");
      failure =0;
      break;
      };

    };

  if (failure==1)
    {puts("No Memory Controller found. Program abort.\n\n\r");
    return;
    };

  mem = (*(pci + id + 0x4)&0xFFFFFFF0); //get BAR0
```

```
help=*(pci + id + 0x1);// enable memory acces
if((help & 0x2)==0)
  {*(pci + id + 0x1)= help | 0x2;
  };


/*get data file*/

puts("Send SREC-File \n\n\r");
first_addr_set = 0;
do {
        gets(r);
        if (r[0] == 'S') {
          addr = h2i(8, &r[4]);
          data = (unsigned char *) &r[12];
          if (r[1] == '3') {
          if(first_addr_set == 0)
            {first_addr = addr;
            first_addr_set =1;
            };
          size = (h2i(2, &r[2])*2 - 10) >> 1;
          last_addr =addr+ size-1;
          for (i=0;i<size;i++) ((char *)addr)[i] = h2i(2,&data[2*i]);
          } else if (r[1] == '7') {
          puts("File received\n\n\r");
          start_addr = addr;
          break;
          }
        }
    } while (1);


/* copy data in memory controller */
puts("Starting copy RAM->Memory Controller\n\n\r");
size = last_addr - first_addr;
size >>=2; //size in words

/*set remap registers*/
*(bridge_cfg+0x3c)=mem;
*(bridge_cfg+0x3d)=0xFFFF0000;
*(bridge_cfg+0x3e)=0x3;

value=(unsigned int *)first_addr;
for(i=0;i<=size;i++)
  {
    *(pci+i)=*(value+i);
  };


/*verify*/
puts("Verifying\n\n\r");

failure = 0;
value=(unsigned int *)first_addr;
for(i=0;i<=size;i++)
  {
    if(*(pci+i)!=*(value+i))
    {failure++;};
    };

puts("Verification completed: ");
if (failure == 0)puts("no errors\n\n\r");
```

```
  else puts(" errors\n\n\r");

}
```

IRQ-Test:

Da IRQs eine Form von Traps darstellen, muss eine Trap-Table benutzt werden. Diese ist in Assembler geschrieben und belegt nach dem Linken den Speicherbereich ab 0x40000000.

```
#define TRAP(H) call H; nop; jmpl %r17, %r0; rett %r18;

.global _trap_table, _start, _skip, _timer1, _timer2, _irq13

_trap_table:
      TRAP(_start)            ;! 00 reset trap
      TRAP(_skip)       ;! 01 instruction_access_exception
      TRAP(_skip)       ;! 02 illegal_instruction
      TRAP(_skip)       ;! 03 priveleged_instruction
      TRAP(_skip)       ;! 04 fp_disabled
      TRAP(_skip)       ;! 05 window_overflow
      TRAP(_skip)       ;! 06 window_underflow
      TRAP(_skip)       ;! 07 memory_address_not_aligned
      TRAP(_skip)       ;! 08 fp_exception
      TRAP(_skip)       ;! 09 data_access_exception
      TRAP(_skip)       ;! 0A tag_overflow
      TRAP(_skip)       ;! 0B watchpoint_exception
      TRAP(_skip)       ;! 0C undefined
      TRAP(_skip)       ;! 0D undefined
      TRAP(_skip)       ;! 0E undefined
      TRAP(_skip)       ;! 0F undefined
      TRAP(_skip)       ;! 10 undefined
      TRAP(_skip)             ;! 11 interrupt level 1
      TRAP(_skip)             ;! 12 interrupt level 2
      TRAP(_skip)             ;! 13 interrupt level 3
      TRAP(_skip)             ;! 14 interrupt level 4
      TRAP(_skip)             ;! 15 interrupt level 5
      TRAP(_skip)             ;! 16 interrupt level 6
      TRAP(_skip)             ;! 17 interrupt level 7
      TRAP(_timer1)             ;! 18 interrupt level 8
      TRAP(_timer2)             ;! 19 interrupt level 9
      TRAP(_skip)             ;! 1A interrupt level 10
      TRAP(_skip)             ;! 1B interrupt level 11
      TRAP(_skip)             ;! 1C interrupt level 12
      TRAP(_irq13)             ;! 1D interrupt level 13
      TRAP(_skip)             ;! 1E interrupt level 14
      TRAP(_skip)             ;! 1F interrupt level 15
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 20 - 23 undefined
      TRAP(_skip)       ;! 24 cp_disabled
      TRAP(_skip);TRAP(_skip);TRAP(_skip);! 25 - 27 undefined
      TRAP(_skip)       ;! 28 cp_exception
      TRAP(_skip);TRAP(_skip);TRAP(_skip);! 29 - 2A undefined
      TRAP(_skip)       ;! 2B data_store_error
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 2C - 2F undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 30 - 33 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 34 - 37 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 38 - 3B undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 3C - 3F undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 40 - 43 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 44 - 47 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 48 - 4B undefined
```

```
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 4C - 4F undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 50 - 53 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 54 - 57 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 58 - 5B undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 5C - 5F undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 60 - 63 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 64 - 67 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 68 - 6B undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 6C - 6F undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 70 - 73 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 74 - 77 undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 78 - 7B undefined
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 7C - 7F undefined

      /* Software traps */
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);!80-84
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 84 - 87
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 88 - 8B
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 8C - 8F
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 90 - 93
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 94 - 97
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 98 - 9B
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! 9C - 9F
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! A0 - A3
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! A4 - A7
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! A8 - AB
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! AC - AF
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! B0 - B3
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! B4 - B7
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! B8 - BB
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! BC - BF
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! C0 - C3
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! C4 - C7
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! C8 - CB
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! CC - CF
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! D0 - D3
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! D4 - D7
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! D8 - DB
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! DC - DF
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! E0 - E3
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! E4 - E7
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! E8 - EB
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! EC - EF
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! F0 - F3
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! F4 - F7
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! F8 - FB
      TRAP(_skip);TRAP(_skip);TRAP(_skip);TRAP(_skip);! FC - FF



_skip:
      nop             ;
      rett            ;
```

Das eigentliche Programm enthält die IRQ-Behandlungs-Routinen:

```
/*************************************************************/
// test of trap-table with IRQs from timer 1, timer 2 and IRQ13
/*************************************************************/

#include "io_leon.h"
```

```
void _irq13()
{struct lregs *regs = (struct lregs *) 0x80000000;
  puts("IRQ13 activated \n\n\r");
}

void _timer1()
{struct lregs *regs = (struct lregs *) 0x80000000;
puts("Timer1-IRQ\n\n\r");
}

void _timer2()
{struct lregs *regs = (struct lregs *) 0x80000000;
puts("Timer2-IRQ\n\n\r");
}


void _start()
{unsigned int i;
 unsigned int IRQ13;
 unsigned int TIMER1;
 unsigned int TIMER2;
 unsigned char pio_byte;

 /* set pio */
 struct lregs *regs = (struct lregs *) 0x80000000;
 i = (regs->piodir );//read pio direction register
 i = i & 0xFFFFFF0F; // pio (4..7) as input
 i = i | 0xC;        // pio (2..3) as output
 regs->piodir = i;   // set direction register

 IRQ13 = 0;
 TIMER1 = 0;
 TIMER2 = 0;

 /* initialize Timers and Scaler*/
 regs->scalerload = 0x000003FF;// 1023
 regs->timerload1 = 0x00011E1A;// 73242
 regs->timerload2 = 0x00023C34;// 146484

 /*set IRQ-Mask register*/
 regs->irqclear = 0x0000FF00; // clear pending interrupts
 regs->irqmask = 0x00002300; // enable IRQ13, 9, 8

 /*enable traps*/
 asm("
set 0x00000000, %o0;
set 0x40000000, %o1;
wr %o0, %o1, %tbr;
rd %psr, %o0;
and %o0, 0xFFFFFFDF, %o0;
wr %o0, 0x20, %psr;
");

do
   {
     for (i=0; i<1000; i++) //wait for some time
       {;};
     pio_byte = regs->piodata;//read Lowest Byte from pio data register


     if (((pio_byte & 0x10) > 1)&& IRQ13 == 0)   //pio(4) enables IRQ13
       {regs->irqforce = 0x00002000;
```

```
      IRQ13 = 1;};
    if ((pio_byte & 0x10) == 0)IRQ13=0;

    if (((pio_byte & 0x20) > 1)&& TIMER1 == 0)   //pio(5) enables TIMER1
      {regs->timerctrl1 = 0x00000007;//reload counter enabled
      TIMER1 = 1;};
    if ((pio_byte & 0x20) == 0)
      {regs->timerctrl1 = 0x00000000;//Timer1 disabled
      TIMER1 =0;};

    if (((pio_byte & 0x40) > 1)&& TIMER2 == 0)   //pio(6) enables TIMER2
      {regs->timerctrl2 = 0x00000007;//reload counter enabled
      TIMER2 = 1;};
    if ((pio_byte & 0x40) == 0)
      {regs->timerctrl2 = 0x00000000;//Timer2 disabled
      TIMER2 =0;};

    if ((pio_byte & 0x80)> 1)// pio(7) as program abort
      {puts("Program aborted by user \n\n\r");

       /*disable traps*/
      asm("
           rd %psr, %o0;
           and %o0, 0xFFFFFFDF, %o0;
           wr %o0, 0x00, %psr;
          ");
      break;};
   }while(1);

};
```